avg = 20.5 stdev = 4.7

1	3	/4
2	2	/8
3	4	/7
4	10	/11

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Fall 2011

Quiz #2: October 14, 2011

Name Michael Plasmeier			*Athena login name* theplaz	*Score* 19
Brad, 26-322 ☐ WF 10 ☐ WF 11	**Silvina**, 34-303 ☐ WF 11 ☐ WF 12	**Li-Shiuan**, 34-304 ☐ WF 12 ☐ WF 1	**Chris**, 34-303 ☒ WF 1 ← I am in this one ☐ WF 2	**David**, 36-155 ☐ WF 2 ☐ WF 3

↑ Not any other one!!! I beg to differ

Notes:

(1) PLEASE FILL IN YOUR NAME, ATHENA NAME, and MARK YOUR SECTION ABOVE before working on the problems on this quiz.

(2) Occasionally, reference materials and/or extra copies of diagrams are provided on the backs of quiz pages. Feel free to use page backs as scratch space, but be sure to mark your answers on the answer lines provided for each question.

Problem 1 (4 points): **Quickies and Trickies**

(A) NovaFlop, Inc advertises a reliable flipflop with an unusual guarantee: it may enter a metastable state if the dynamic discipline is not followed, but that when the metastable state settles to a valid logic level, it will always be a 1 rather than a 0.

last year's exam

Is their claim plausible? Circle one: YES … NO … Can't Tell

(B) MetaSure, Inc advertises a 2-input device that claims to produce a positive transition on its output within 1 ns after asynchronous positive transitions have occurred on both of its inputs.

Is their claim plausible? Circle one: YES … NO … Can't Tell

(C) An FSM, M, is constructed using a 7-state FSM S and a 5-state FSM F as components. The only two inputs to F are two of the three state bits of S, the single input to S serves as the only input to M, and F's single output is the only output of M. What can you say about the maximum number of states that M might exhibit?

M: S (7) → F (5)

Maximum number of states for M (or "can't tell"): 35 states

can't reduce at all w/ # of inputs ← can't tell

(D) A synchronous pipeline has a latency of L and throughput of T. If we find a way to double the clock frequency by doubling the number of pipeline stages, what revised latency and throughput can we expect?

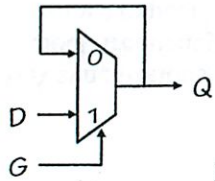
New Latency: 2L ; **Throughput:** 2T (−1)

stages · tpd max stage ex $\frac{1}{10} \to \frac{1}{5}$ is 2T

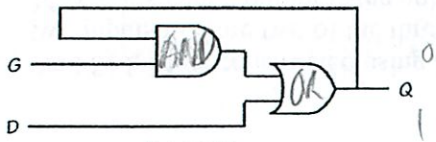
Starts w/ 0

(Scratch copies of diagrams)

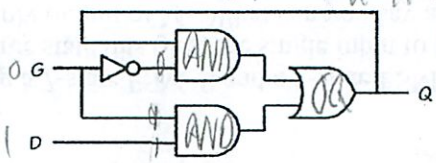
DG	Mux Q	A Q	B Q	C Q
00	0	0	0	
01	0	0	0	
10	0	1	0	
11	1	1	1	1



00
01
10
11



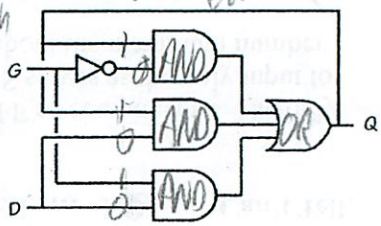
Proposal A



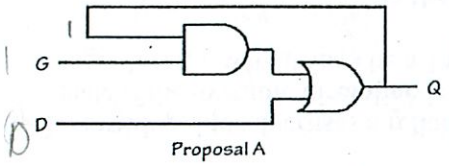
Proposal B

AND AND - Not a latch

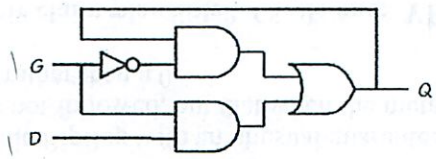
Not a latch either



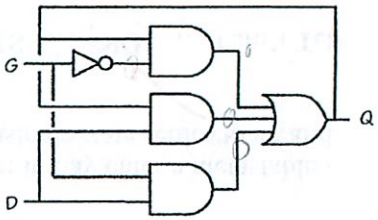
Proposal C



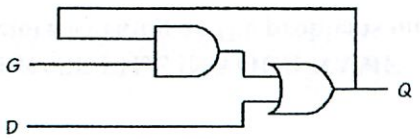
Proposal A



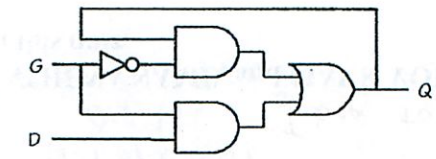
Proposal B



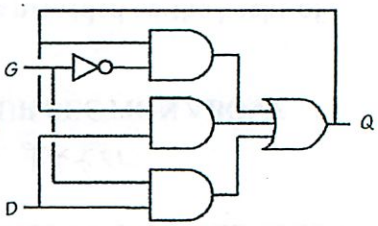
Proposal C



Proposal A



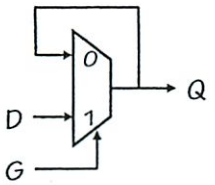
Proposal B



Proposal C

Problem 2 (8 points): **Latches Revisited**

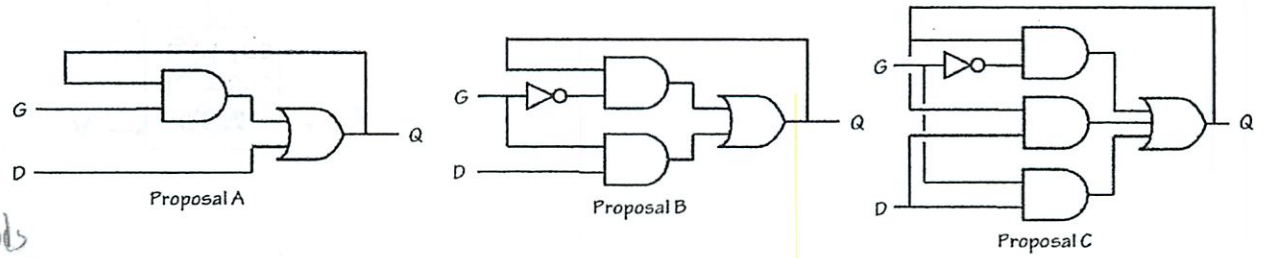
Untel, Inc is a startup exploring a new gate technology that has hired you as a consultant. They have learned how to make reliable, lenient AND gates, OR gates, and inverters, but don't yet have a cell library offering devices like multiplexors. Their current crisis, for which they need your help, is the design of a reliable latch.



The Untel engineers vaguely remember a 6.004 lecture showing how to make a latch using a lenient multiplexor (as shown to the left), and reason that they can make a latch at least as good starting from AND/OR/inverter logic.

Umm never a good assumption

There are three different proposals being considered:



try methods

The Untel CTO shows you the diagrams, and asks you characterize each as

- BAD, meaning it doesn't work reliably;
- GOOD, meaning that it works reliably (given appropriate dynamic discipline rules); or
- OBESE, meaning that it works but uses more gates than necessary.

(A) (6 points) Characterize each of the above proposals.

-4

Proposal A (circle one):	**BAD** ...	**GOOD** ...	**OBESE**
Proposal B (circle one):	**BAD** ...	**GOOD** ...	**OBESE**
Proposal C (circle one):	**BAD** ...	**GOOD** ...	**OBESE**

The Untel engineers listen carefully to your advice, and eventually design a latch design along with an informal argument (similar to that given in lecture) that the latch works properly. The remaining project is to specify minimal setup and hold times for the latch that guarantee its proper operation. While the analysis in lecture derived these specifications from the propagation delay of the multiplexor used there to build the latch, the Untel engineers must use the propagation delays of the AND/OR/invert gates, each of which is 1 nanosecond. Assume the contamination delay of the gates is zero.

(B) **(2 points)** Give appropriate setup and hold time specifications for a latch built using the above components. [HINT: This requires careful analysis!]

-2

G, so 1 must work!

not thinking right abt clk

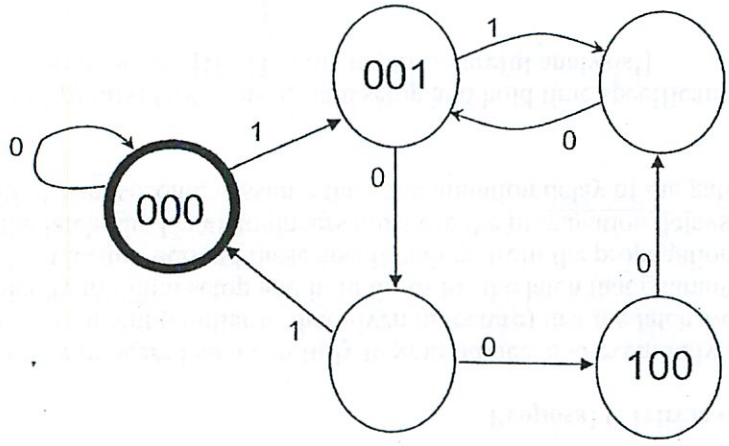
-when does it matter

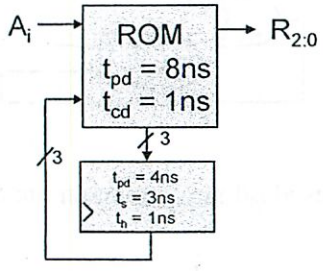
clk when valid at!

Setup time: 3 ns

Hold time: 1 ns

(Scratch copies of diagrams)





"whats mod 5 of a binary #"

32 16 8 4 2 1

Problem 3 (7 points): **High Fives**

The Professor who supervises your UROP project has designed an FSM that determines the remainder modulo 5 of an arbitrary binary number. The FSM has a three-bit output, $R_2R_1R_0$, which is initially 000. As successive bits of the number are entered left-to-right, the outputs change to reflect the modulo 5 value of the number entered thus far. For example, if the input sequence were 010101, the outputs observed in consecutive clock cycles would be 000, 000, 001, 010, 000, 000, 001.

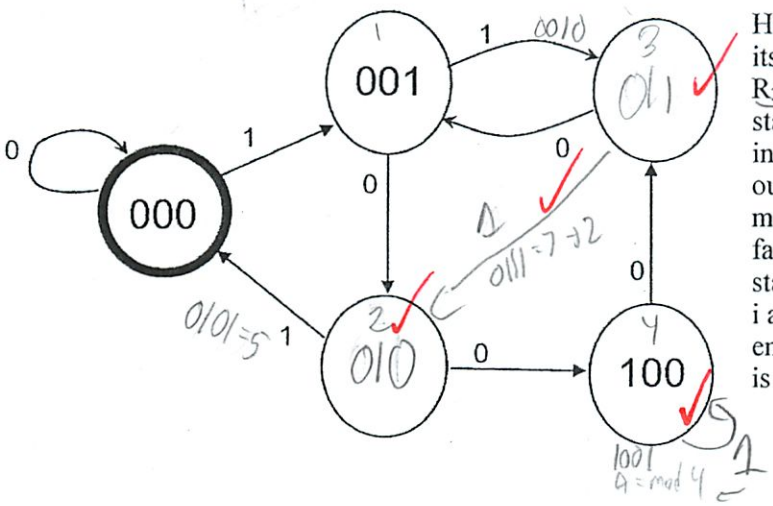
start

Outputs

4 1 = 5 mod 5 = 0
8 2 10 mod 5 = 0

The Professor's design is shown to the left. Of course, since he's a Professor, it contains no errors. However, his absent mindedness ("I've got a mind like a steel whatchamacallit") has led to omission of some important detail.

add 1. 0111 = 7 = 2



He has cleverly designed the FSM so that its three state bits are precisely the $R_2R_1R_0$ output bits, and has marked most states with their outputs. Note that the initial state, S0, is marked with the outputs 000. and is the state of the machine when the number entered thus far is zero mod 5. Each of the other states Si has the three-bit binary value of i as its output, and corresponds to the entry of a number whose value modulo 5 is i.

(A) **(4 points)** Complete the above diagram by filling in the two missing output values and the two missing state transitions. Be sure to mark each of the transitions you add with an appropriate input value causing that transition.

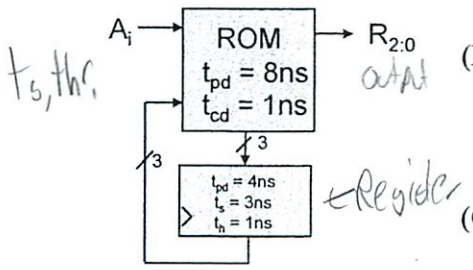
weird problem!

(Complete above diagram)

You are asked to implement the FSM using the diagram shown to the left, involving a ROM having propagation and contamination delays of 8ns and 1ns, and a register whose propagation and contamination delays are 4ns and zero, and whose setup and hold times are 3ns and 1ns, respectively.

$t_h \le t_{cd}$

$t_{clk} \ge \Sigma t_{pd} + t_s$



(B) **(1 point)** What is the minimum clock period that can be applied to this FSM?

$t_{clk} \ge \Sigma t_{pd} + t_s$ before register

8 + 3

Minimum clock period: 11 ✗ ns

-3

(C) **(2 points)** What is the setup and hold time requirements on the input to this FSM?

edit I copy right in notes!

$t_h \le t_{cd}$

$t_{clk} \ge \Sigma t_p + t_s$

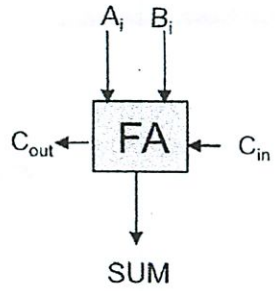
$11 \ge 8 + 3$

Setup time: 3 ✗ ns

Hold time: 1 ✗ ns

0/2

Problem 4 (11 points): **Addplex, Inc**

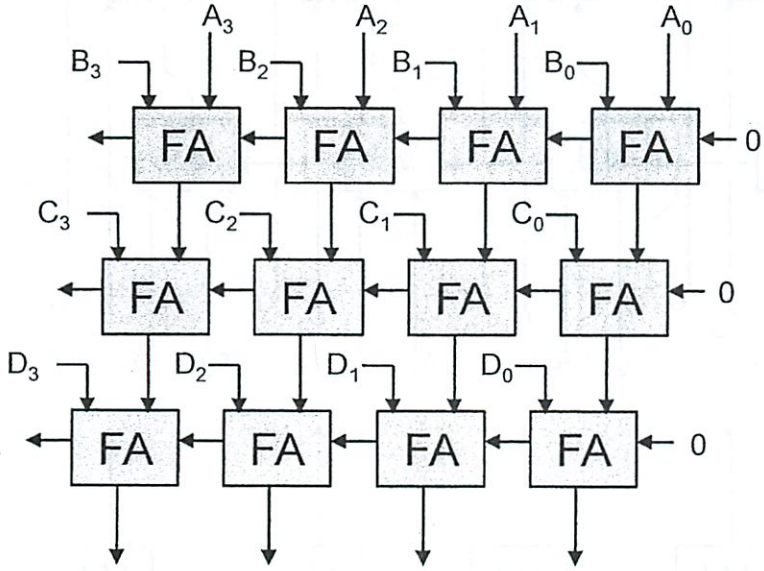


You've been hired by Addplex, a startup specializing in adders capable of summing an entire column of binary numbers as a single operation. Until you came on board, they had no engineers, but had a number of customers for combinational devices capable of adding N k-bit numbers together for various values of N and k. You need to quickly design products to satisfy this demand. You remember the full adder device you designed in lab 2 (symbol shown to the right); fortunately, Addplex has a large inventory of these on hand.

N=4 k=4

You begin by tackling the problem of designing combinational circuit that adds four 4-bit binary numbers, producing a 4-bit binary result. Your approach is to combine three copies of the 4-bit adder you built in lab 2. and combine them as shown below:

A, B, C, D



You **connect the low-order carry inputs to 0, and simply ignore the high-order carry outputs.**

Note that the four 4-bit inputs are designated A[3:0], B[3:0], C[3:0], and D[3:0].

You learn that the propagation delay of each full adder module is 1 nanosecond.

(A) **(1 point)** What is the propagation delay of the above adder?

Σt_{pd} along critical (max) path

Propagation delay of above adder: 6 ✓ **ns**

$(k-1)+(N-2)+1$

You consider generalizing this approach to add N k-bit binary numbers for arbitrary values of N and k. You are particularly interested in the asymptotic cost and performance characteristics of the adder as N and k become large, and recall the Θ(...) notation that abstracts out additive and multiplicative constants.

(B) **(2 points)** Using Θ(...) notation, give asymptotic latency and hardware cost of an adder constructed as above capable of adding N k-bit quantities. Give simplest possible expressions in N and k.

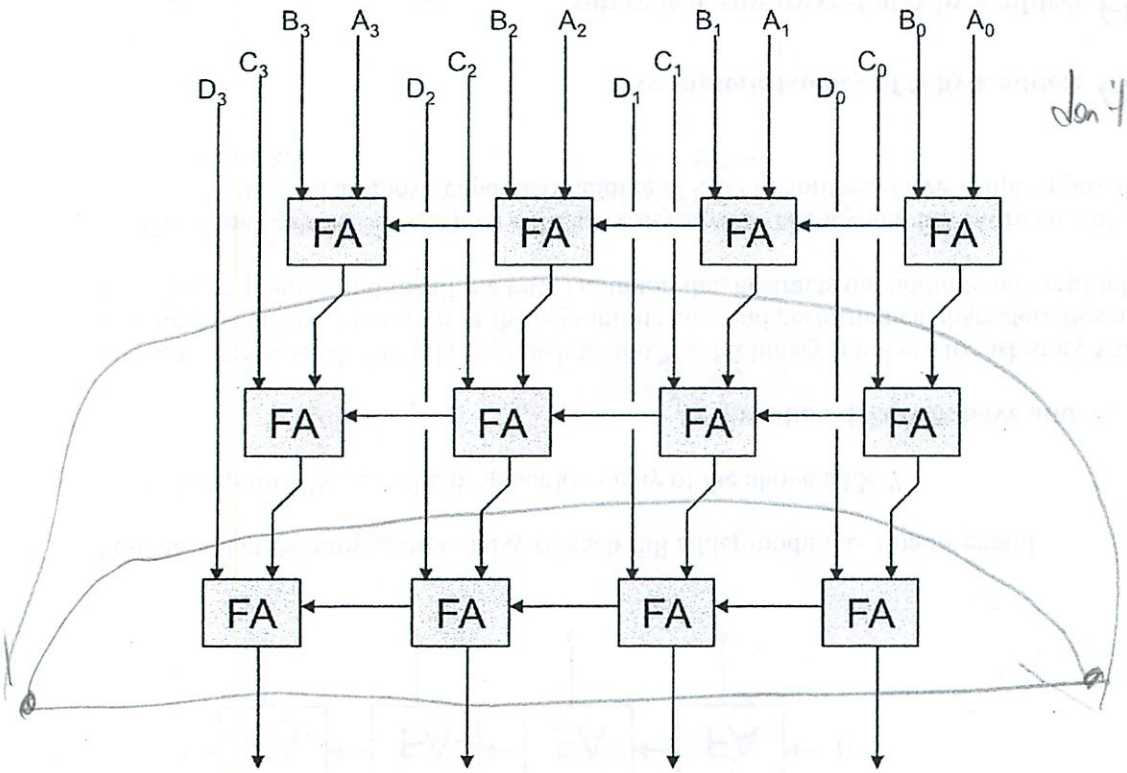
Asymptotic latency of N by k adder: Θ(N+k ✓) add 2 for each

t_{pd}

Asymptotic hardware cost of N by k adder: Θ(N·k ✓)

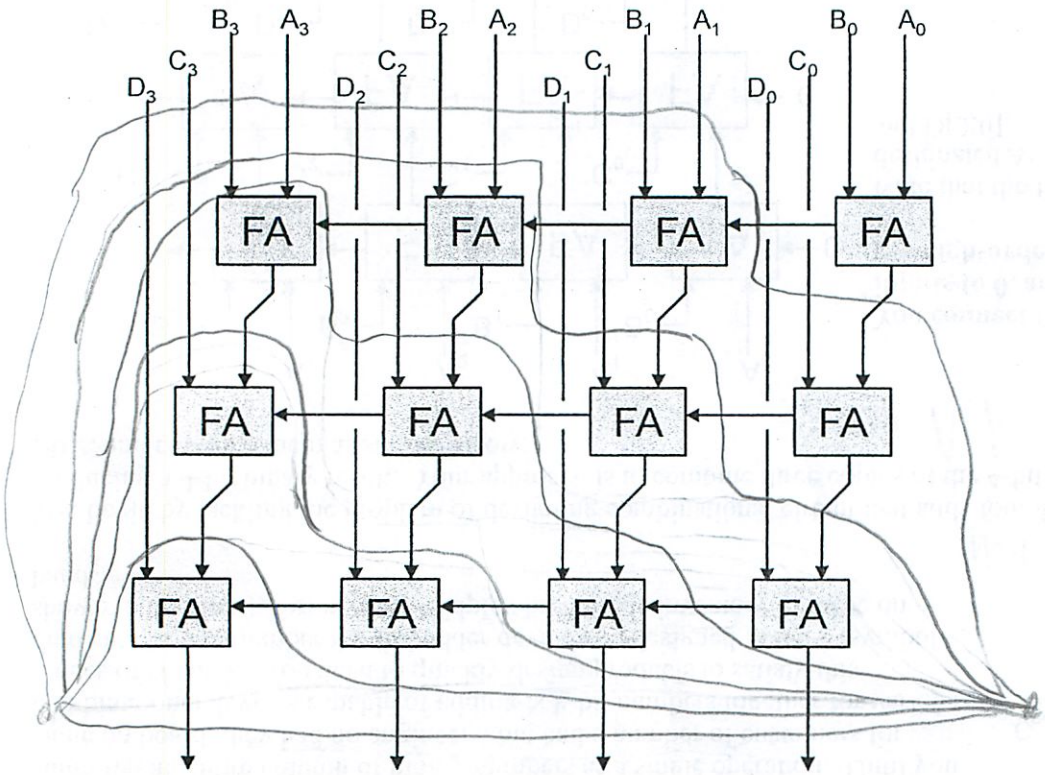
$(N-1)\cdot k$

(Scratch copies of diagrams)

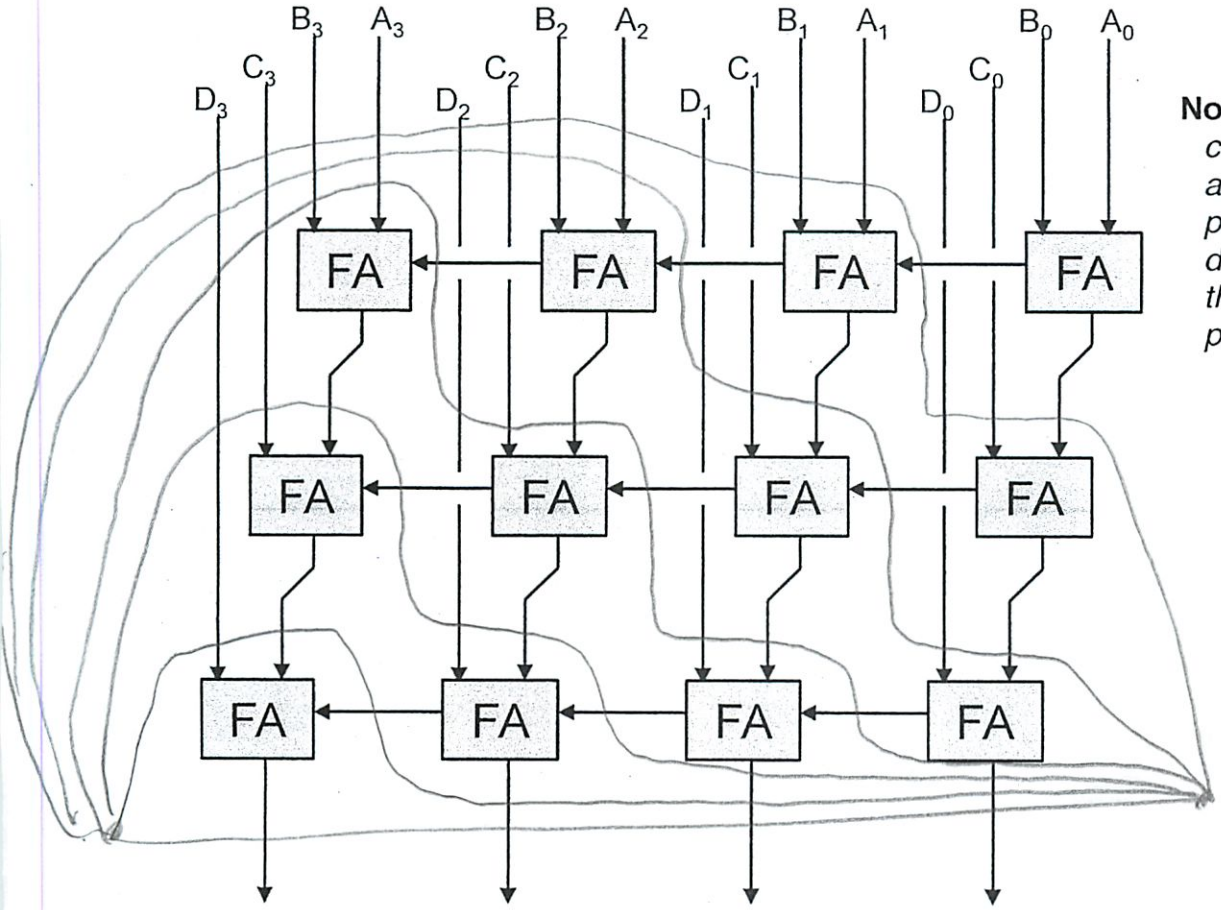


don't need input

Can still be 4
- need weird diagonal



Next, you turn to the problem of pipelining your adders for maximum throughput. You start by pipelining your 4x4 combinational adder, using ideal (zero-delay) registers inserted at strategic positions:



Note: *spare (scratch) copies of this diagram appear on the back of the previous page. Please debug your answer using the scratch copies, and present it neatly here.*

(C) **(5 points)** Indicate, on the above diagram, register locations for a maximum-throughput pipelined implementation of the 4 by 4 adder. You may indicate locations by drawing contours marking appropriate pipeline stage boundaries. Use the minimum number of registers necessary to maximize throughput; remember to put registers on all outputs.

(mark register locations on the above diagram)

Finally, consider generalizing the pipelined adder to add N k-bit numbers.

(D) **(3 points)** Using Θ(...) notation, give asymptotic latency, throughput, and hardware cost of a pipelined adder capable of adding N k-bit quantities. Give simplest possible expressions in N and k.

Asymptotic latency of N by k adder: Θ(k+N) ← 6·1 k+N · T_{PD}

Asymptotic throughput of N by k adder: Θ(1) 1/T_{PD}

Asymptotic hardware cost of N by k adder: Θ((k·N)²) (−1) only core boxes

k·N · k·N · k·N
before #lines reg/line

END OF QUIZ!

1st Done 15 min

1	/ 4
2	/ 8
3	/ 7
4	/ 11

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Fall 2011

Quiz #2: October 14, 2011

Name Solutions			*Athena login name*	*Score*
Brad, 26-322 ☐ WF 10 ☐ WF 11	**Silvina**, 34-303 ☐ WF 11 ☐ WF 12	**Li-Shiuan**, 34-304 ☐ WF 12 ☐ WF 1	**Chris**, 34-303 ☐ WF 1 ☐ WF 2	**David**, 36-155 ☐ WF 2 ☐ WF 3

Notes:

(1) PLEASE FILL IN YOUR NAME, ATHENA NAME, and MARK YOUR SECTION ABOVE before working on the problems on this quiz.

(2) Occasionally, reference materials and/or extra copies of diagrams are provided on the backs of quiz pages. Feel free to use page backs as scratch space, but be sure to mark your answers on the answer lines provided for each question.

Problem 1 (4 points): **Quickies and Trickies**

(A) NovaFlop, Inc advertises a reliable flipflop with an unusual guarantee: it may enter a metastable state if the dynamic discipline is not followed, but that when the metastable state settles to a valid logic level, it will always be a 1 rather than a 0.

Is their claim plausible? Circle one: YES ... NO ... Can't Tell

(B) MetaSure, Inc advertises a 2-input device that claims to produce a positive transition on its output within 1 ns after asynchronous positive transitions have occurred on both of its inputs.

Is their claim plausible? Circle one: YES ... NO ... Can't Tell

(C) An FSM, M, is constructed using a 7-state FSM S and a 5-state FSM F as components. The only two inputs to F are two of the three state bits of S, the single input to S serves as the only input to M, and F's single output is the only output of M. What can you say about the maximum number of states that M might exhibit?

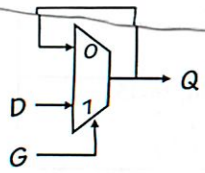
Maximum number of states for M (or "can't tell"): 35

(D) A synchronous pipeline has a latency of L and throughput of T. If we find a way to double the clock frequency by doubling the number of pipeline stages, what revised latency and throughput can we expect?

New Latency: L **; Throughput:** 2*T

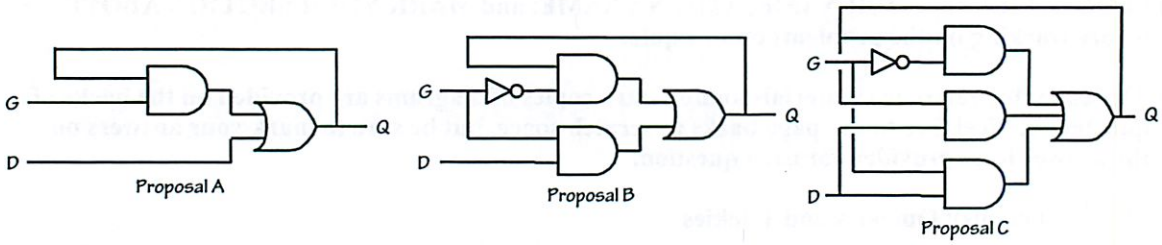
Problem 2 (8 points): **Latches Revisited**

Untel, Inc is a startup exploring a new gate technology that has hired you as a consultant. They have learned how to make reliable, lenient AND gates, OR gates, and inverters, but don't yet have a cell library offering devices like multiplexors. Their current crisis, for which they need your help, is the design of a reliable latch.



The Untel engineers vaguely remember a 6.004 lecture showing how to make a latch using a lenient multiplexor (as shown to the right), and reason that they can make a latch at least as good starting from AND/OR/inverter logic.

There are three different proposals being considered:



The Untel CTO shows you the diagrams, and asks you characterize each as

- BAD, meaning it doesn't work reliably;
- GOOD, meaning that it works reliably (given appropriate dynamic discipline rules); or
- OBESE, meaning that it works but uses more gates than necessary.

(A) (6 points) Characterize each of the above proposals.

A is just nonsense.
B uses a non-lenient MUX.
C uses a lenient MUX, and works.

This was a disaster qv

Proposal A (circle one): **BAD** ... **GOOD** ... **OBESE**

Proposal B (circle one): **BAD** ... **GOOD** ... **OBESE**

Proposal C (circle one): **BAD** ... **GOOD** ... **OBESE**

The Untel engineers listen carefully to your advice, and eventually design a latch design along with an informal argument (similar to that given in lecture) that the latch works properly. The remaining project is to specify minimal setup and hold times for the latch that guarantee its proper operation. While the analysis in lecture derived these specifications from the propagation delay of the multiplexor used there to build the latch, the Untel engineers must use the propagation delays of the AND/OR/invert gates, each of which is 1 nanosecond. Assume the contamination delay of the gates is zero.

(B) **(2 points)** Give appropriate setup and hold time specifications for a latch built using the above components. [HINT: This requires careful analysis!]

Looking at Proposal C:

G=1: D must force 2 inputs of lenient OR to be stable & valid for 1ns to guarantee OR's output when G is changed.

Setup time: 4 **ns**

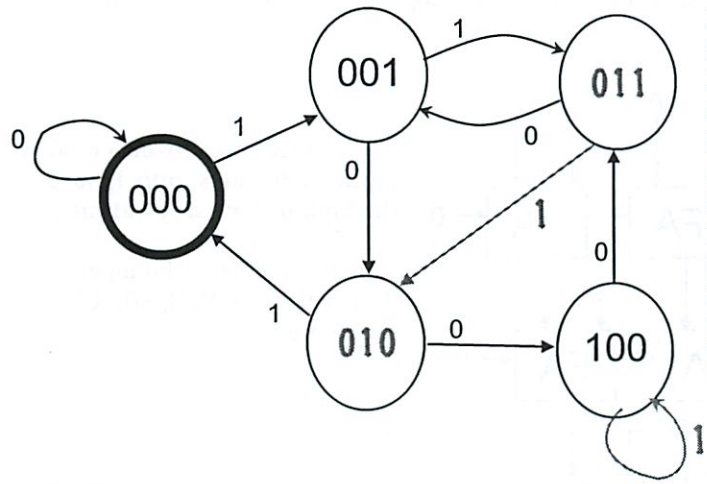
G=0: Cannot change D until GD value has provided a stable input to the OR for 1 ns

Hold time: 3 **ns**

Problem 3 (7 points): **High Fives**

The Professor who supervises your UROP project has designed an FSM that determines the remainder modulo 5 of an arbitrary binary number. The FSM has a three-bit output, $R_2R_1R_0$, which is initially 000. As successive bits of the number are entered left-to-right, the outputs change to reflect the modulo 5 value of the number entered thus far. For example, if the input sequence were 010101, the outputs observed in consecutive clock cycles would be 000, 000, 001, 010, 000, 000, 001.

The Professor's design is shown to the left. Of course, since he's a Professor, it contains no errors. However, his absent mindedness ("I've got a mind like a steel whatchamacallit") has led to omission of some important detail.

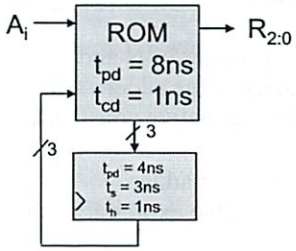


He has cleverly designed the FSM so that its three state bits are precisely the $R_2R_1R_0$ output bits, and has marked most states with their outputs. Note that the initial state, S0, is marked with the outputs 000. and is the state of the machine when the number entered thus far is zero mod 5. Each of the other states Si has the three-bit binary value of i as its output, and corresponds to the entry of a number whose value modulo 5 is i.

(A) **(4 points)** Complete the above diagram by filling in the two missing output values and the two missing state transitions. Be sure to mark each of the transitions you add with an appropriate input value causing that transition.

(Complete above diagram)

You are asked to implement the FSM using the diagram shown to the left, involving a ROM having propagation and contamination delays of 8ns and 1ns, and a register whose propagation delay is 4ns and whose setup and hold times are 3ns and 1ns, respectively.



(B) **(1 point)** What is the minimum clock period that can be applied to this FSM?

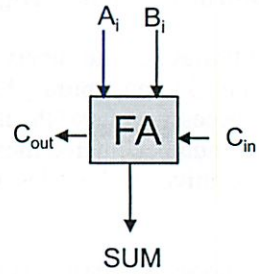
Minimum clock period: 15 **ns**

(C) **(2 points)** What is the setup and hold time requirements on the input to this FSM?

Setup time: 11 **ns**

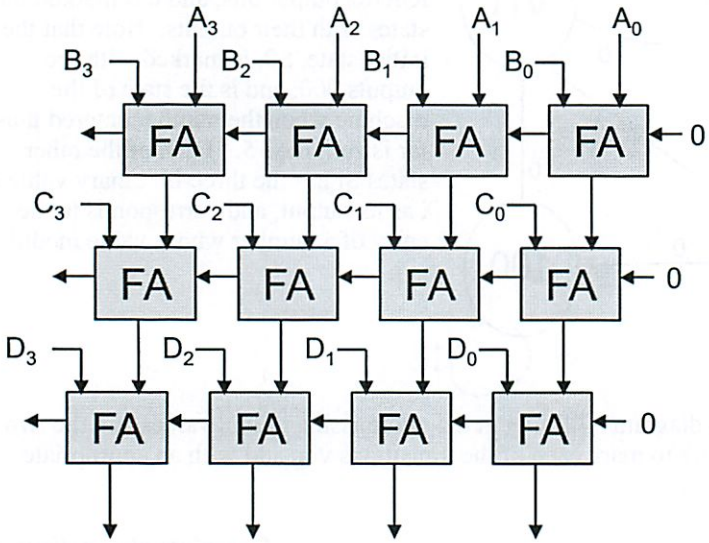
Hold time: 0 **ns**

Problem 4 (11 points): **Addplex, Inc**



You've been hired by Addplex, a startup specializing in adders capable of summing an entire column of binary numbers as a single operation. Until you came on board, they had no engineers, but had a number of customers for combinational devices capable of adding N k-bit numbers together for various values of N and k. You need to quickly design products to satisfy this demand. You remember the full adder device you designed in lab 2 (symbol shown to the right); fortunately, Addplex has a large inventory of these on hand.

You begin by tackling the problem of designing combinational circuit that adds four 4-bit binary numbers, producing a 4-bit binary result. Your approach is to combine three copies of the 4-bit adder you built in lab 2. and combine them as shown below:



You **connect the low-order carry inputs to 0, and simply ignore the high-order carry outputs.**

Note that the four 4-bit inputs are designated A[3:0], B[3:0], C[3:0], and D[3:0].

You learn that the propagation delay of each full adder module is 1 nanosecond.

(A) **(1 point)** What is the propagation delay of the above adder?

Propagation delay of above adder: 6 **ns**

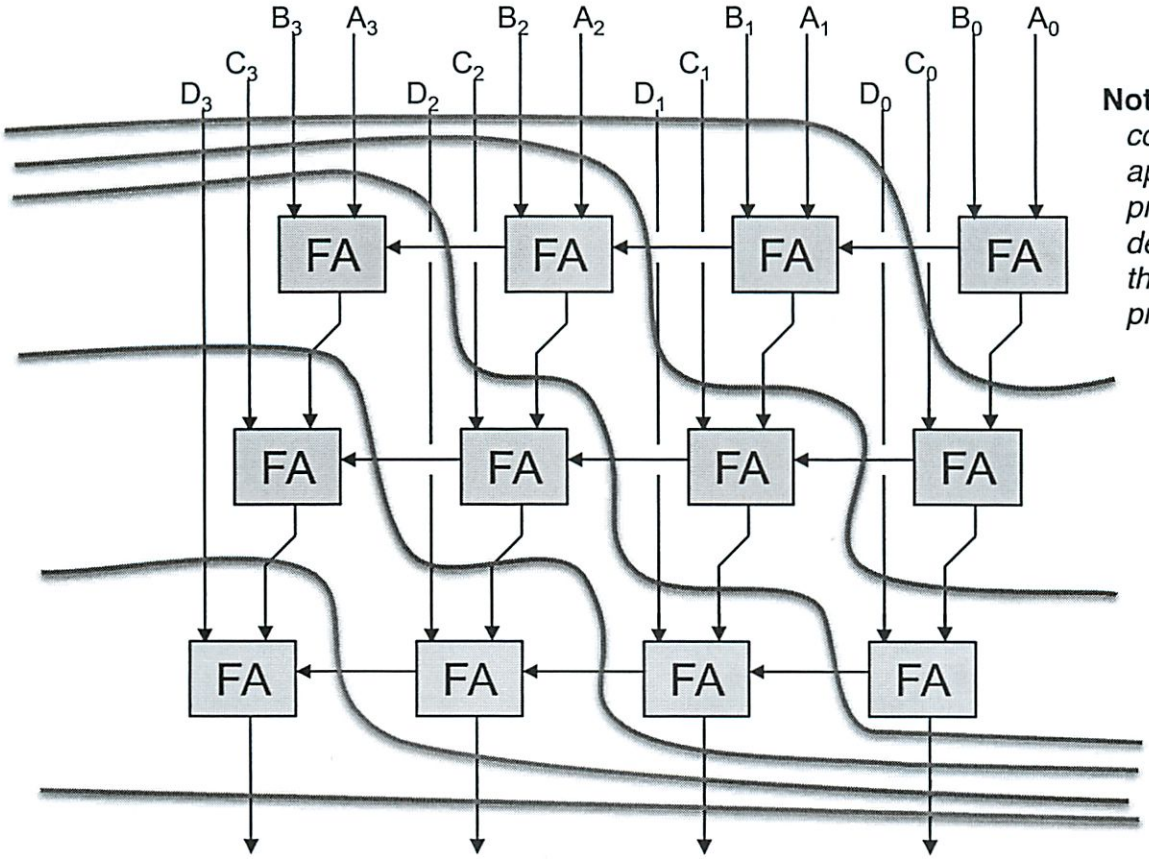
You consider generalizing this approach to add N k-bit binary numbers for arbitrary values of N and k. You are particularly interested in the asymptotic cost and performance characteristics of the adder as N and k become large, and recall the Θ(...) notation that abstracts out additive and multiplicative constants.

(B) **(2 points)** Using Θ(...) notation, give asymptotic latency and hardware cost of an adder constructed as above capable of adding N k-bit quantities. Give simplest possible expressions in N and k.

Asymptotic latency of N by k adder: Θ(N * k)

Asymptotic hardware cost of N by k adder: Θ(N * k)

Next, you turn to the problem of pipelining your adders for maximum throughput. You start by pipelining your 4x4 combinational adder, using ideal (zero-delay) registers inserted at strategic positions:



Note: *spare (scratch) copies of this diagram appear on the back of the previous page. Please debug your answer using the scratch copies, and present it neatly here.*

(C) **(5 points)** Indicate, on the above diagram, register locations for a maximum-throughput pipelined implementation of the 4 by 4 adder. You may indicate locations by drawing contours marking appropriate pipeline stage boundaries. Use the minimum number of registers necessary to maximize throughput; remember to put registers on all outputs.

(mark register locations on the above diagram)

Finally, consider generalizing the pipelined adder to add N k-bit numbers.

(D) **(3 points)** Using Θ(...) notation, give asymptotic latency, throughput, and hardware cost of a pipelined adder capable of adding N k-bit quantities. Give simplest possible expressions in N and k.

Asymptotic latency of N by k adder: Θ(N + k)

Asymptotic throughput of N by k adder: Θ(1)

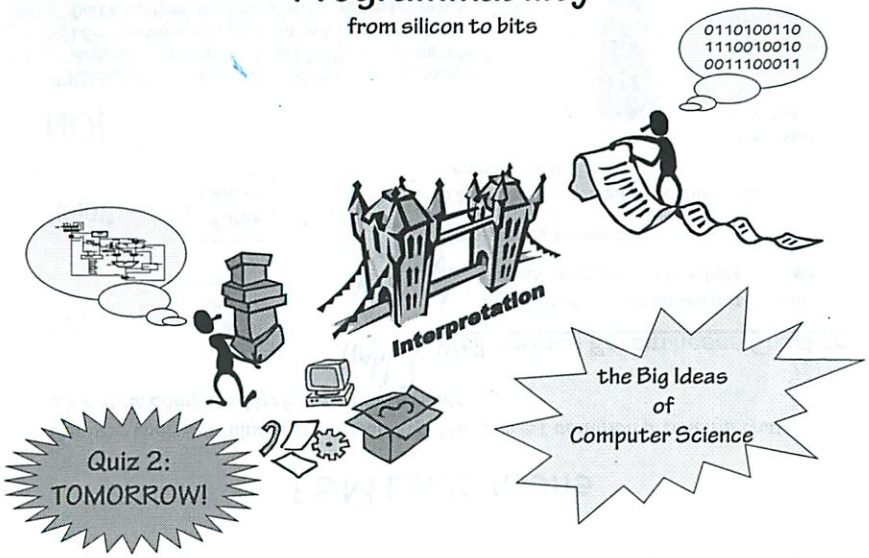
Asymptotic hardware cost of N by k adder: Θ(N * k)

END OF QUIZ!

10/13

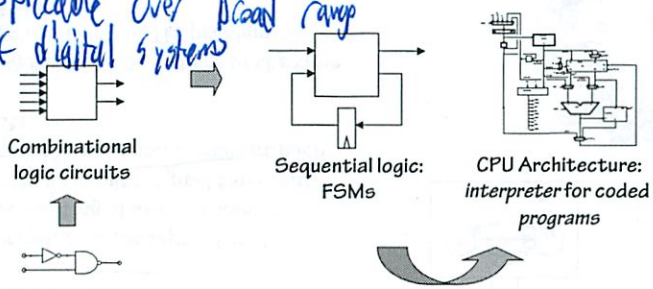
Programmability

from silicon to bits



6.004 Roadmap

applicable over broad range of digital systems



Now: narrow focus to general purpose computers

Programmability: Models

- Interpretation; Programs; Languages; Translation
- Beta implementation
- Pipelined Beta
- Software conventions
- Memory architectures

did bottom up approach

FSMs as Programmable Machines

ROM-based FSM sketch:

in, state, out bits

Given i, s, and o, we need a ROM organized as:

2^{i+s} words x (o+s) bits

So how many possible i-input, o-output, FSMs with s-state bits, exist?

~~possible~~ cols/ cols

$$2^{(o+s)2^{i+s}}$$

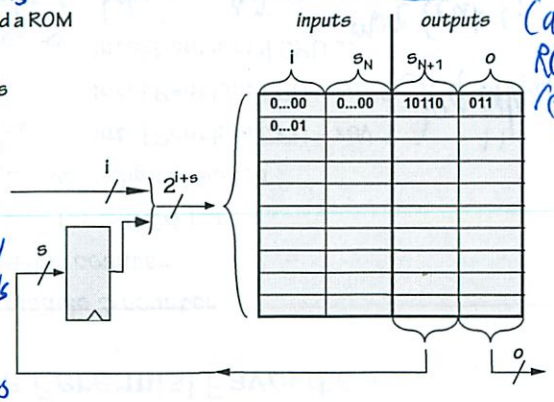
(some may be equivalent)

rows

can be 1 or 0

An FSM's behavior is completely determined by its ROM contents.

Can program ROM to represent



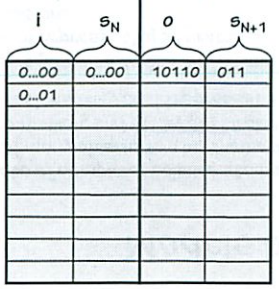
always finite might be large

Big Idea #1:

FSM Enumeration

GOAL: List all possible FSMs in some canonical order.

- INFINITE list, but
- Every FSM has an entry and an associated index.



i	s	o	FSM#	Truth Table
1	1	1	1	00000000
1	1	1	2	00000001
			...	...
1	1	1	256	11111111
2	2	2	257	000000...000000
2	2	2	258	000000...000001
			...	...
3	3	3		000000...000000
			...	
4	4	4		000000...000000

2^8 FSMs

2^{64}

List well defined

What if s=2, i=o=1??

Every possible FSM can be associated with a number. We can discuss the ith FSM

can ignore 2nd in and 2nd out

5/3

Some Perennial Favorites...

FSM_{837}	modulo 3 counter
FSM_{1077}	4-bit counter
FSM_{1537}	lock for 6.004 Lab
FSM_{89143}	Steve's digital watch
$FSM_{22698469884}$	Intel Pentium CPU – rev 1
$FSM_{784362783}$	Intel Pentium CPU – rev 2
$FSM_{72698436563783}$	Intel Pentium II CPU

equivalent

more features

↑ need to define behavior out of each FSM

Reality: The integer indexes of actual FSMs are *much bigger* than the examples above. They must include enough information to constitute a *complete description* of each device's unique structure.

Models of Computation

before computers

The roots of computer science stem from the study of many alternative mathematical "models" of computation, and ~~study of the classes of~~ computations they could represent.

An elusive goal was to find an "ultimate" model, capable of representing all practical computations...

- switches
- gates
- combinational logic
- memories
- FSMs

Can every computer be represented as a FSM?

No!

We've got FSMs ... what else do we need?

Are FSMs the ultimate digital computing device?

FSM Limitations

Despite their usefulness and flexibility, there exist common problems that cannot be computed by FSMs. For instance:

"(()())" → Paren Checker → OK

"(())())" → Paren Checker → Nix

light it

well balanced

Well-formed Parentheses Checker:

Given any string of coded left & right parens, outputs 1 if it is balanced, else 0.

Simple, easy to describe.

Is this device equivalent to one of our enumerated FSMs???

NO!

PROBLEM: Requires ARBITRARILY many states, depending on input. Must "COUNT" unmatched LEFT parens. An FSM can only keep track of a finite number of unmatched parens: for every FSM, we can find a string it can't check.

talked about in recitation

I know how to fix that!

Alan Turing

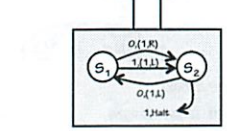
bug in FSM is finiteness of memory

Big Idea #2:

Turing Machines

loops around!

Alan Turing was one of a group of researchers studying alternative models of computation.



He proposed a conceptual model consisting of an FSM combined with an infinite digital tape that could be read and written at each step.

Turing's model (like others of the time) solves the "FINITE" problem of FSMs.

This baby won't run out of memory!

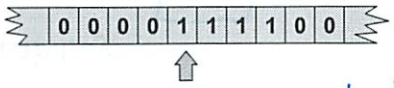
A Turing machine Example

Turing Machine Specification

- Doubly-infinite tape
- Discrete symbol positions
- Finite alphabet – say {0, 1}
- Control FSM
  - INPUTS:
    - Current symbol
  - OUTPUTS:
    - write 0/1
    - move Left/Right
- Initial Starting State {S0}
- Halt State {Halt}

A Turing machine, like an FSM, can be specified with a truth table. The following Turing Machine implements a unary (base 1) incrementer.

Current State	Input	Next State	Write Tape	Move Tape
S0	1	S0	1	R
S0	0	S1	1	L
S1	1	S1	1	L
S1	0	HALT	0	R

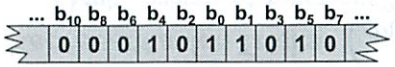


OK, but how about *real* computations... like fact(n)?

Underwhelmed!

Turing Machine Tapes as Integers

Canonical names for bounded tape configurations:



Encode tape position

FSM_{347}

That's just Turing Machine 347 operating on tape 51

– take 1 bit from left + right

Encoding: starting at current position, build a binary integer taking successively higher-order bits from right and left sides. If nonzero region is bounded, eventually all 1's will be incorporated into the resulting integer representation.

– can convert to ∞ long #

What calculations be performed here?

TMs as Integer Functions

Want it to embody some integer function

Turing Machine T_i operating on Tape x, where $x = \ldots b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

$y = T_i[x]$

x: input tape configuration
y: output tape configuration

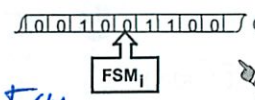
I wonder if a TM can compute *EVERY* integer function...

read off output tape value

Meanwhile, Turing's buddies were busy too...

Alternative models of computation

Turing Machines [Turing]



Turing

link b/w FSMs and TMs

Recursive Functions [Kleene]

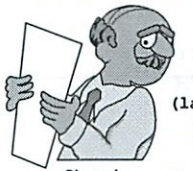
$F(0,x) = x$

$F(1+y,x) = 1+F(x,y)$

```
(define (fact n)
  (... (fact (- n 1)) ...))
```

Kleene

Lambda calculus [Church, Curry, Rosser...]



Church

$\lambda x.\lambda y.xxy$

```
(lambda (x) (lambda (y) (x (x y))))
```

↑ scans on lisp community influential in programming

Production Systems [Post, Markov]

Post

$\alpha \rightarrow \beta$

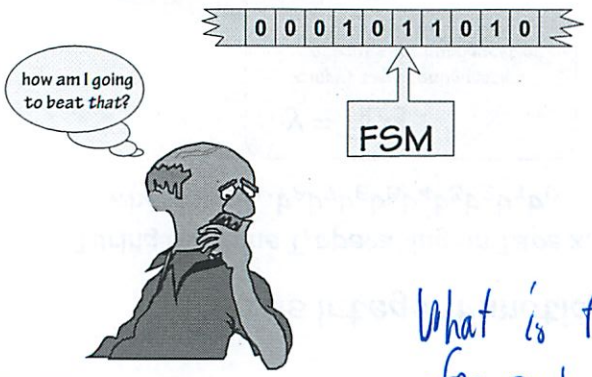
```
IF pulse=0 THEN
   patient=dead
```

AI expert systems

The 1st Computer Industry Shakeout

Competing models of Computation

Here's a TM that computes SQUARE ROOT!



What is the set of fns for each model

And the battles raged

Here's a Lambda Expression that does the same thing...

(λ(x) .....)

... and here's one that computes the nth root for ANY n!

(λ(x n) .....)

maybe if I gave away a microwave oven with every Turing Machine...

decades before Computers

CONTEST: Which model computes more functions?

RESULT: an N-way TIE!

Big Idea #3:

Computability

FACT: Each model studied is capable of computing exactly the same set of integer functions!

Proof Technique:
Constructions that translate between models

BIG IDEA:
Computability, independent of computation scheme chosen

unproved, but universally accepted...

Church's Thesis:

Every discrete function computable by ANY realizable machine is computable by some Turing machine.

No one can prove or figure out how to

if by Turing machine

Computable Functions

f(x) *computable* <=> for some k, all x:
$f(x) = T_K[x] \equiv f_K(x)$

Equivalently: f(x) computable on Cray, Pentium, in C, Python, Java, ...

Representation Tricks: Can extend easily to ...

- Multi-argument functions? to compute $f_k(x,y)$, use
  <x,y> = integer whose *even* bits come from x, and whose *odd* bits come from y; whence
  $f_K(x, y) = T_K[<x, y>]$
- Data types: Can encode characters, strings, floats, ... as integers.
- Alphabet size: use groups of N bits for 2^N symbols

Enumeration of Computable functions

Conceptual table of ALL Turing Machine behaviors...

VERTICAL AXIS: Enumeration of TMs (computable functions)

HORIZONTAL AXIS: Enumeration of input tapes.

ENTRY AT (n, m): Result of applying m^{th} TM to argument n

INTEGER k: TM halts, leaving k on tape.

★ : TM never halts.

Sometimes TM may never halt

f_i	$f_i(0)$	$f_i(1)$	$f_i(2)$	...	$f_i(n)$	...
f_0	37	23	★	...	33	...
f_1	42	★	111	...	12	...
f_2	★	★	★	...	★	...
...	...	...	...	...	...	...
f_m	0	★	831	...	$f_m(n)$	...
...	...	...	...	...	...	...

∞ table

aren't *all* well-defined integer functions computable?

NO!

there are simply too many integer functions to fit in our enumeration!

Can this table define every integer fn ∈ no. of int fns

Halting function

Uncomputable Functions

Unfortunately, not every well-defined integer function is computable. The most famous such function is the so-called Halting function, $f_H(k, j)$, defined by:

$$f_H(k,j) = \begin{cases} 1 & \text{if } T_k[j] \text{ halts;} \\ 0 & \text{otherwise.} \end{cases}$$

$f_H(k, j)$ determines whether the kth TM halts when given a tape containing j.

THEOREM: f_H is different from every function in our enumeration of computable functions; hence it cannot be computed by any Turing Machine.

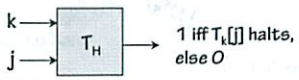
PROOF TECHNIQUE: "Diagonalization" (after Cantor, Gödel)

- If f_H is computable, it is equivalent to some TM (say, T_H).
- Using T_H as a component, we can construct another TM whose behavior differs from every entry in our enumeration and hence must not be computable.
- Hence f_H cannot be computable.

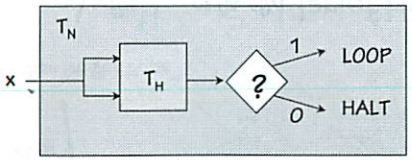
Simple proof

Why f_H is uncomputable

If f_H is computable, it is equivalent to some TM (say, T_H):



Then T_N (N for "Nasty"), which must be computable if T_H is:



$T_N[x]$: LOOPS if $T_x[x]$ halts; HALTS if $T_x[x]$ loops

if one thing computable, other thing is

Finally, consider giving N as an argument to T_N:

$T_N[N]$: LOOPS if $T_N[N]$ halts; HALTS if $T_N[N]$ loops

Contradiction!

T_N can't be computable, hence T_H can't either!

Footnote: Diagonalization

(clever proof technique used by Cantor, Gödel, Turing)

If T_H exists, we can use it to construct T_N. Hence T_N is computable if T_H is. (informally we argue by Church's Thesis; but we can show the actual T_N construction, if pressed)

Set theory behind everything

Why T_N can't be computable:

f_i	$f_i(0)$	$f_i(1)$	$f_i(2)$	...	$f_i(n)$	...
f_0	★	23	★	...	33	...
f_1	42	★	111	...	12	...
f_2	★	★	★	...	★	...
...	...	...	...	★	...	...
f_m	0	★	831	...	$f_m(n)$	...
...	...	...	...	...	...	...

T_N differs from every computable function for at least one argument – along the diagonal of our table. Hence T_N can't be among the entries in our table!

Hence no such T_H can be constructed, even in theory.

Computable Functions: A TINY SUBSET of all Integer functions!

Brief History of Mathematics (6.004 view)

GP Computers

Doom. MP3s. Windows. Payroll accounting. Blue screen crashes. 6.004.

"bliss"

{It's all { sets }}

"yeah? then I'm the Pope"
- Russell

Math = Bunk???

"OK, here's the program"
- Hilbert

Search for "perfect" logic: consistent, complete

"This statement can't be proved"
- Gödel

{ consistency, completeness }...
... take your pick

"Some things just don't compute..."
- Turing

computability

"All you need, in theory..."
- Turing

universality

"Let's build this baby..."
- von Neumann

von Neumann architecture

one inconsistency and you can prove anything

earlier repaired 6.042

repair math

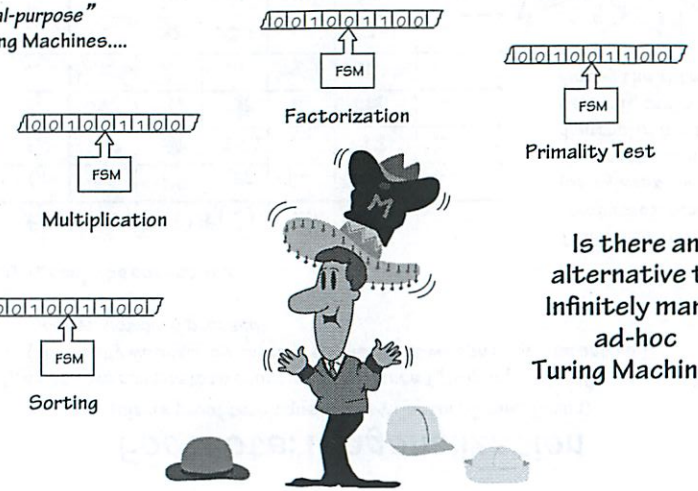
people depressed

at Princeton Univ Adv Study

meanwhile...

Turing machines Galore!

"special-purpose" Turing Machines....



Is there an alternative to Infinitely many, ad-hoc Turing Machines?

lots of stuff solved w/ Turing machine

The Universal Function

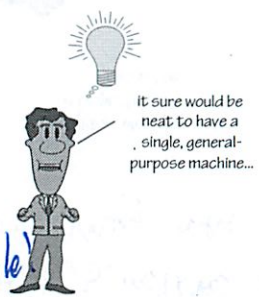
OK, so there are uncomputable functions – infinitely many of them, in fact.

Here's an interesting candidate to explore: the Universal function, U, defined by

$$U(k, j) = T_k[j]$$

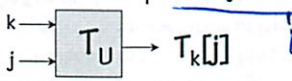
machine tape

Could this be computable???



would not need a warehouse of TMs if had this

SURPRISE! U is computable by a Turing Machine:

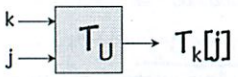


is Computable

In fact, there are infinitely many such machines. Each is capable of performing *any* computation that can be performed by *any* TM!

Big Idea #4:

Universality



What's going on here?

k encodes a "program" – a description of some arbitrary machine.

j encodes the input data to be used.

T_U *interprets* the program, emulating its processing of the data!

KEY IDEA: Interpretation. Manipulate *coded representations* of computing machines, rather than the machines themselves.

how can we build such a machine?

magic threshold for a GP computer

Turing Universality: The *Universal Turing Machine* is the paradigm for modern general-purpose computers! (cf: earlier *special-purpose* computers)

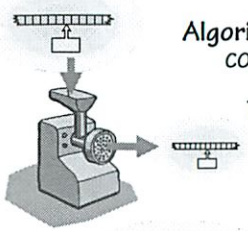
- Basic threshold test: Is your machine *Turing Universal*? If so, it can emulate every other Turing machine!
- Remarkably low threshold: UTMs with handfuls of states exist.
- Every modern computer is a UTM (given enough memory)
- To show your machine is Universal: demonstrate that it can emulate some known UTM.

first few were very complex, then simpler

Launched CS

Coded Algorithms: Key to CS

data vs hardware

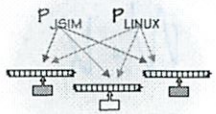


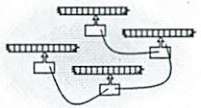
Can't input 1 machine into another

Algorithms as data: enables
COMPILERS: analyze, optimize, transform behavior

$T_{COMPILER\text{-}X\text{-}to\text{-}Y}[P_X] = P_Y$, such that
$T_X[P_X, z] = T_Y[P_Y, z]$

but compilers do this!





SOFTWARE ENGINEERING:
Composition, iteration, abstraction of coded behavior
$F(x) = g(h(x), p((q(x)))$

LANGUAGE DESIGN: Separate specification from implementation
- C, Java, JSIM, Linux, ... all run on X86, Sun, ARM, JVM, CLR, ...
- Parallel development paths:
  - Language/Software design
  - Interpreter/Hardware design

theoritical analysis

Summary

Formal models (computability, Turing Machines, Universality) provide the basis for modern computer science:
- Fundamental limits (what can't be done, even given plenty of memory and time)
- Fundamental equivalence of computation models
- Representation of algorithms as data, rather than machinery
- Programs, Software, Interpreters, Compilers, ...

They leave many practical dimensions to deal with:
- Costs: Memory size, Time Performance
- Programmability

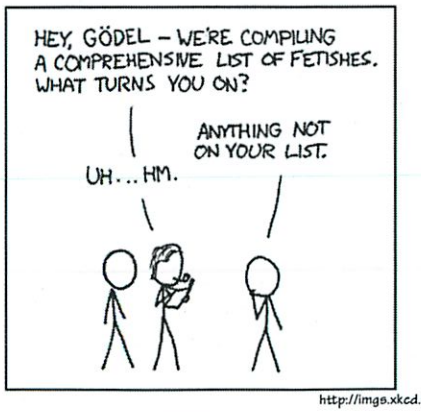
Next step: Design of a practical interpreter!

epilogue I:

The Gates Paradox

AUTHOR KATHARINE GATES RECENTLY ATTEMPTED TO MAKE A CHART OF ALL SEXUAL FETISHES.

LITTLE DID SHE KNOW THAT RUSSELL AND WHITEHEAD HAD ALREADY FAILED AT THIS SAME TASK.



http://imgs.xkcd.com/comics/fetishes.png

epilogue II:

Gödel's Incompleteness Theorem

explained in words of one syllable

First of all, when I say "proved", what I will mean is "proved with the aid of the whole of math". Now then: two plus two is four, as you well know. And, of course, it can be proved that two plus two is four (proved, that is, with the aid of the whole of math, as I said, though in the case of two plus two, of course we do not need the whole of math to prove that it is four). And, as may not be quite so clear, it can be proved that it can be proved that two plus two is four, as well. And it can be proved that it can be proved that it can be proved that two plus two is four. And so on. In fact, if a claim can be proved, then it can be proved that the claim can be proved. And that too can be proved.

Now: two plus two is not five. And it can be proved that two plus two is not five. And it can be proved that it can be proved that two plus two is not five, and so on.

Thus: it can be proved that two plus two is not five. Can it be proved as well that two plus two is five? It would be a real blow to math, to say the least, if it could. If it could be proved that two plus two is five, then it could be proved that five is not five, and then there would be no claim that could not be proved, and math would be a lot of bunk.

So, we now want to ask, can it be proved that it can't be proved that two plus two is five? Here's the shock: no, it can't. Or to hedge a bit: if it can be proved that it can't be proved that two plus two is five, then it can be proved as well that two plus two is five, and math is a lot of bunk. In fact, if math is not a lot of bunk, then no claim of the form "claim X can't be proved" can be proved.

So, if math is not a lot of bunk, then, though it can't be proved that two plus two is five, it can't be proved that it can't be proved that two plus two is five.

By the way, in case you'd like to know: yes, it can be proved that if it can be proved that it can't be proved that two plus two is five, then it can be proved that two plus two is five.

George Boolos, 1994

all true

Hint: If I were taking test tomorrow
last thing I would write is N.k

10/18

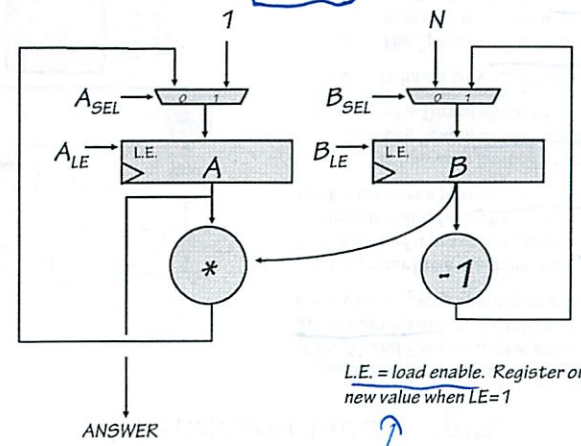
Designing an Instruction Set

10/18
Skipped Class
– review later

Lab 4 due Thursday!

Let's Build a Simple Computer

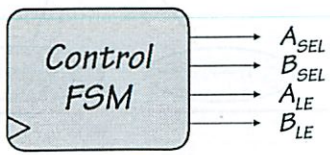
Data path for computing N*(N-1)

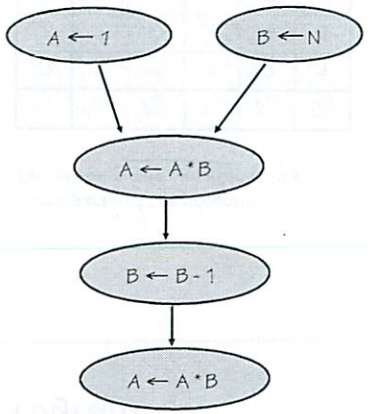


L.E. = load enable. Register only loads new value when LE=1

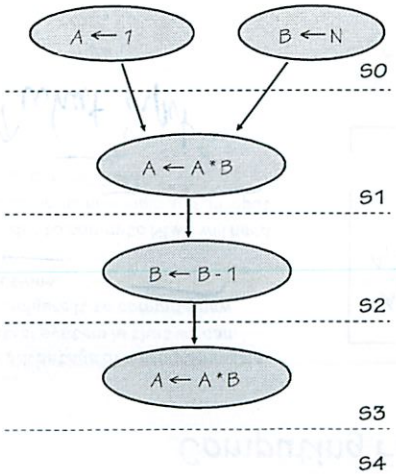
A Programmable Control System

Computing N*(N-1) with this data path is a multi-step process. We can control the processing at each step with an FSM. If we allow different control sequences to be loaded into the control FSM, then we allow the machine to be programmed.





A First Program



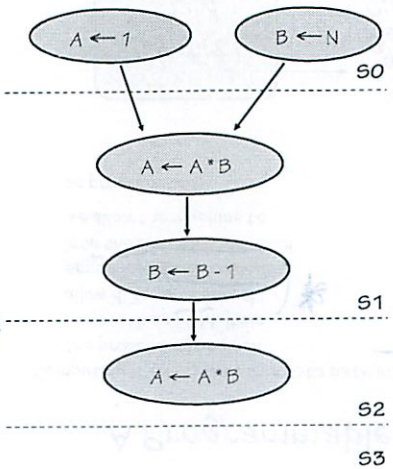
Once more, writing a control program is nothing more than filling in a table:

S_N	S_{N+1}	A_{sel}	A_{LE}	B_{sel}	B_{LE}
0	1	1	1	0	1
1	2	0	1	0	0
2	3	0	0	1	1
3	4	0	1	0	0
4	4	0	0	0	0

like a Turing machine

10/18

An Optimized Program



why can we do this?

Some parts of the program can be computed simultaneously:

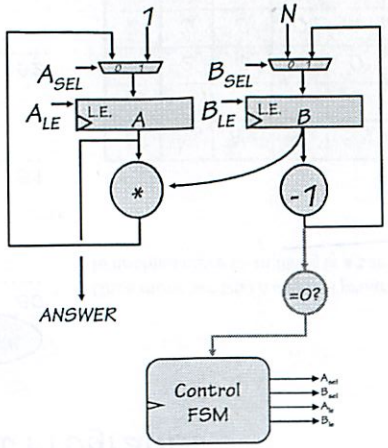
S_N	S_{N+1}	A_{sel}	A_{LE}	B_{sel}	B_{LE}
0	1	1	1	0	1
1	2	0	1	1	1
2	3	0	1	0	0
3	3	0	0	0	0

Computing Factorial

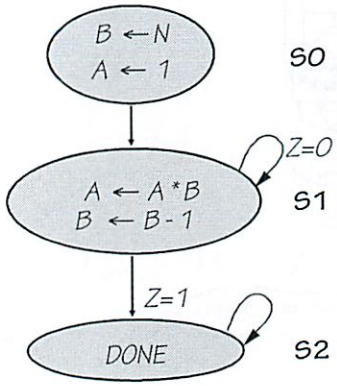
The advantage of a programmable control system is that we can reconfigure it to compute new functions.

In order to compute N! we will need to add some new logic and an input to our control FSM:

↑ what input?



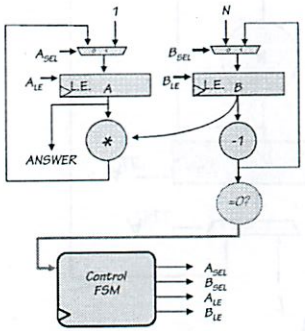
Control Structure for Factorial



Programmability allows us to reuse data paths to solve new problems. What we need is a general purpose data path, which can be used to efficiently solve most problems as well as an easier way to control it.

Z	S_N	S_{N+1}	A_{sel}	A_{LE}	B_{sel}	B_{LE}
-	0	1	1	1	0	1
0	1	1	0	1	1	1
1	1	2	0	1	1	1
-	2	2	0	0	0	0

A Programmable Engine



We've used the same data paths for computing N*(N-1) and Factorial; there are a variety of other computations we might implement simply by re-programming the control FSM.

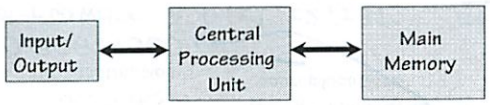
Although our little machine is programmable, it falls short of a practical general-purpose computer – and fails the Turing Universality test – for three primary reasons:

1. It has very limited storage: it lacks the "expandable" memory resource of a Turing Machine.
2. It has a tiny repertoire of operations.
3. The "program" is fixed. It lacks the power, e.g., to generate a new program and then execute it.

A General-Purpose Computer

The von Neumann Model

Many architectural approaches to the general purpose computer have been explored. The one on which nearly all modern, practical computers is based was proposed by John von Neumann in the late 1940s. Its major components are:



Central Processing Unit (CPU): containing several registers, as well as logic for performing a specified set of operations on their contents.

Memory: storage of N *words* of W bits each, where W is a fixed architectural parameter, and N can be expanded to meet needs.

I/O: Devices for communicating with the outside world.

The Stored Program Computer

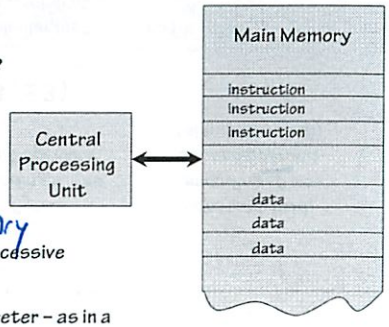
The von Neumann architecture easily addresses the first two limitations of our simple programmable machine example:

- A richer repertoire of operations, and
- An expandable memory.

But how does it achieve *programmability*?

Key idea: Memory holds not only data, but *coded instructions* that make up a *program*.

↑ instructions in memory

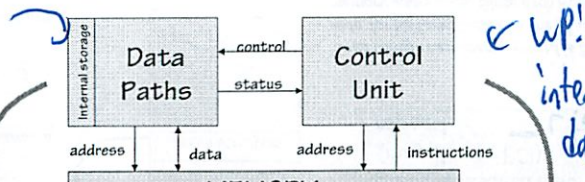


CPU fetches and executes – interprets – successive instructions of the program ...

- Program is simply data for the interpreter – as in a Universal Turing Machine!
- Single expandable resource pool – main memory – constrains both data and program size.

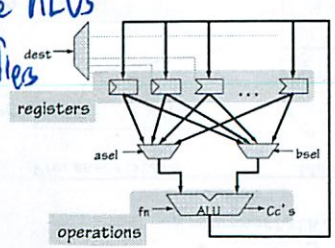
Anatomy of a von Neumann Computer

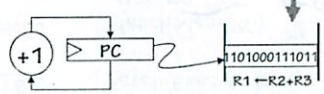
What are data paths.



← WP! Controls interaction b/w datapath + memory

WP: functional units like ALUs or multipliers





- INSTRUCTIONS *coded as binary data*
- PROGRAM COUNTER or PC: Address of next instruction to be executed
- logic to translate instructions into control signals for data path

Instruction Set Architecture

Coding of instructions raises some interesting choices...

- Tradeoffs: performance, compactness, programmability
- Uniformity. Should different instructions
  - Be the same size?
  - Take the same amount of time to execute?
    - ➢ Trend: Uniformity. Affords simplicity, speed, pipelining.
- Complexity. How many different instructions? What level operations?
  - Level of support for particular software operations: array indexing, procedure calls, "polynomial evaluate", etc
    - ➢ "Reduced Instruction Set Computer" (RISC) philosophy: simple instructions, optimized for speed

↑ only have simple instructions

Mix of engineering & Art...

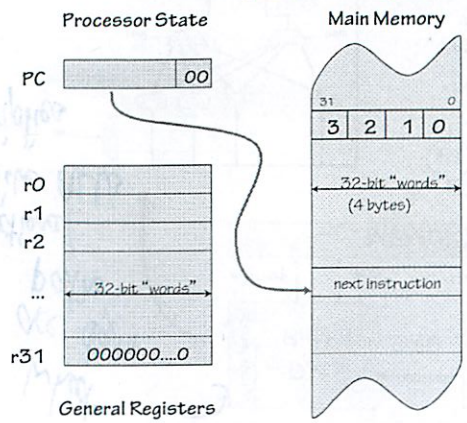
Trial (by simulation) is our best technique for making choices!

Our representative example: the β architecture!

↑ the 6.004 architecture

β Programming Model

a representative, simple, contemporary RISC



Fetch/Execute loop:

- fetch Mem[PC]
- PC = PC + 4† ← state
- execute fetched instruction (may change PC!)
- repeat! ← to imp

†Even though each memory word is 32-bits wide, for historical reasons the β uses byte memory addresses. Since each word contains four 8-bit bytes, addresses of consecutive words differ by 4.

β Instruction Formats

All Beta instructions fit in a single 32-bit word, whose fields encode combinations of

- a 6-bit OPCODE (specifying one of < 64 operations)
- several 5-bit OPERAND locations, each one of the 32 registers
- an embedded 16-bit constant ("literal")

There are two instruction formats:

- Opcode, 3 register operands (2 sources, destination)

OPCODE	r_c	r_a	r_b	*unused*

- Opcode, 2 register operands, 16-bit literal constant

OPCODE	r_c	r_a	16-bit signed constant

so how does this work on hardware

β ALU Operations

Sample coded operation: ADD instruction

100000	00011	00001	00010	0000000000000000
				unused

OPCODE = 100000, encoding ADD

r_c=3, encoding R3 as destination

r_a=1, r_b=2 encoding R1 and R2 as source locations

encode it to hex

32-bit hex: 0x80611000

What *we* prefer to write: `ADD(r1,r2,r3)` "assembly language" oh that is what that is

`ADD(ra,rb,rc)`:

Reg[rc] ← Reg[ra] + Reg[rb]

"Add the contents of ra to the contents of rb; store the result in rc"

Similar instructions for other ALU operations:

arithmetic: ADD, SUB, MUL, DIV
compare: CMPEQ, CMPLT, CMPLE
boolean: AND, OR, XOR, XNOR
shift: SHL, SHR, SAR

β ALU Operations with Constant

ADDC instruction: adds constant, register contents:

110000	00011	00001	1111111111111101

OPCODE = 110000, encoding ADDC

r_c=3, encoding R3 as destination

r_a=1, encoding R1 as first operand

constant field, encoding -3 as second operand (sign-extended!)

Symbolic version: `ADDC(r1,-3,r3)`

`ADDC(ra,const,rc)`:

Reg[rc] ← Reg[ra] + sxt(const)

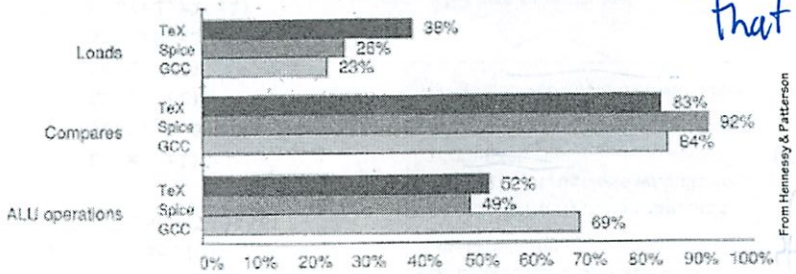
"Add the contents of ra to const; store the result in rc"

Similar instructions for other ALU operations:

arithmetic: ADDC, SUBC, MULC, DIVC
compare: CMPEQC, CMPLTC, CMPLEC
boolean: ANDC, ORC, XORC, XNORC
shift: SHLC, SHRC, SARC

Do We Need Built-in Constants?

how would that help?



Percentage of the operations that use a constant operand

> One way to answer architectural questions is to evaluate the consequences of different choices using carefully chosen representative benchmarks (programs and/or code sequences). Make choices that are "best" according to some metric (cost, performance, ...).

Engineering!

Baby's First Beta Program

(fragment)

Suppose we have N in r1, and want to compute N*(N-1), leaving the result in r2:

Sub 1 from r1 Put in r2

```
SUBC(r1,1,r2)     | put N-1 into r2
MUL(r2,r1,r2)     | leave N*(N-1) in r2
```

multiply r2·r1 → r2

These two instructions do what our little ad-hoc machine did. Of course, limiting ourselves to registers for storage falls short of our ambitions.... it amounts to the finite storage limitations of an FSM!

Needed: instruction-set support for reading and writing locations in main memory...

β Loads & Stores

OPCODE	r_c	r_a	16-bit signed constant

Load

`LD(ra,const,rc)` Reg[rc]←Mem[Reg[ra] + sxt(const)] (address)

"Fetch into rc the contents of the memory location whose address is C plus the contents of ra"

Abbreviation: `LD(C,rc)` for `LD(R31,C,rc)`

Store

`ST(rc,const,ra)` Mem[Reg[ra] + sxt(const)]←Reg[rc]

"Store the contents of rc into the memory location whose address is C plus the contents of ra"

Abbreviation: `ST(rc,C)` for `ST(rc,C,R31)`

> BYTE ADDRESSES, but only 32-bit word accesses to word-aligned addresses are supported. Low two address bits are ignored!

??

r31 = empty?

so must be in memory first

Storage Conventions

- Variables live in memory (Addr assigned at compile time)
- Operations done on registers
- Registers hold Temporary values

how get stuff in here

```
int x, y;
y = x * 37;
```

Compilation approach: LOAD, COMPUTE, STORE

translates to

```
LD(r31, 0x1008, r0)
MULC(r0, 37, r0)
ST(r0, 0x100C, r31)
```

← from x1008 load into reg r0

1000:	n
1004:	r
1008:	x
100C:	y
1010:	

or, more humanely, to

```
x=0x1008
y=0x100C
LD(x, r0)
MULC(r0, 37, r0)
ST(r0, y)
```

put as variables

Ra defaults to R31 (0)

I don't get diff memory and registers

Common "Addressing Modes"

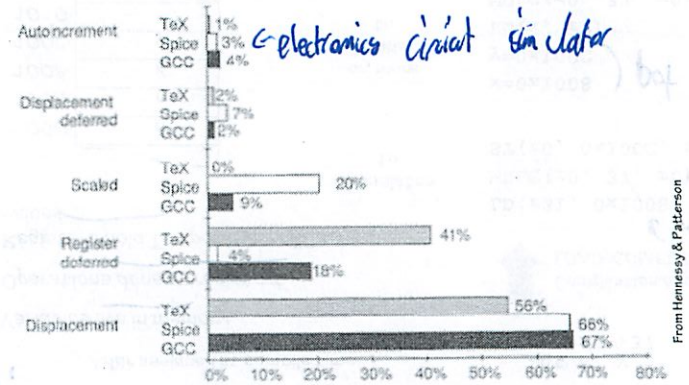
β can do these with appropriate choices for Ra and const

- Absolute: "constant"
  - Value = Mem[constant]
  - Use: accessing static data
- Indirect (aka Register deferred): "(Rx)"
  - Value = Mem[Reg[x]]
  - Use: pointer accesses
- Displacement: "constant(Rx)"
  - Value = Mem[Reg[x] + constant]
  - Use: access to local variables
- Indexed: "(Rx + Ry)"
  - Value = Mem[Reg[x] + Reg[y]]
  - Use: array accesses (base+index)
- Memory indirect: "@(Rx)"
  - Value = Mem[Mem[Reg[x]]]
  - Use: access thru pointer in mem
- Autoincrement: "(Rx)+"
  - Value = Mem[Reg[x]]; Reg[x]++
  - Use: sequential pointer accesses
- Autodecrement: "-(Rx)"
  - Value = Reg[X]--; Mem[Reg[x]]
  - Use: stack operations
- Scaled: "constant(Rx)[Ry]"
  - Value = Mem[Reg[x] + c + d*Reg[y]]
  - Use: array accesses (base+index)

Argh! Is the complexity worth the cost?
Need a cost/benefit analysis!

think more about how this makes of pointers

Memory Operands: Usage



← electronics circuit simulator

Usage of different memory operand modes

Capability so far: Expression Evaluation

Translation of an Expression:

```
int x, y;
y = (x-3)*(y+123456)

x:      long(0)
y:      long(0)
c:      long(123456)
   ...
   LD(x, r1)
   SUBC(r1,3,r1)
   LD(y, r2)
   LD(c, r3)
   ADD(r2,r3,r2)
   MUL(r2,r1,r1)
   ST(r1,y)
```

- VARIABLES are allocated storage in main memory
- VARIABLE references translate to LD or ST
- OPERATORS translate to ALU instructions
- SMALL CONSTANTS translate to ALU instructions w/ built-in constant
- "LARGE" CONSTANTS translate to initialized variables

Yeah where store variable reference?

NB: Here we assume that variable addresses fit into 16-bit constants!

Can We Run *Every* Algorithm?

Model thus far:

- Executes instructions sequentially –
- Number of operations executed = number of instructions in our program!

Good news: programs can't "loop forever"!

- Halting problem is solvable for our current Beta subset!

NOT Universal!

Bad news: can't compute Factorial:

- Only supports bounded-time computations;
- Can't do a loop, e.g. for Factorial!

Needed: ability to change the PC.

Low/why?

Beta Branch Instructions

The Beta's *branch instructions* provide a way of conditionally changing the PC to point to some nearby location... allowing loops

... and, optionally, remembering (in Rc) where we came from (useful for procedure calls).

OPCODE	r_c	r_a	16-bit signed constant

NB: "offset" is a SIGNED CONSTANT encoded as part of the instruction!

`BEQ(ra,label,rc)`: Branch if equal

```
PC = PC + 4;
Reg[rc] = PC;
if (REG[ra] == 0)
   PC = PC + 4*offset;
```

`BNE(ra,label,rc)`: Branch if not equal

```
PC = PC + 4;
Reg[rc] = PC;
if (REG[ra] != 0)
   PC = PC + 4*offset;
```

```
offset = (label - <addr of BNE/BEQ>)/4 - 1
       = up to 32767 instructions before/after BNE/BEQ
```


Now we can do Factorial...

Synopsis (in C):

- Input in n, output in ans
- r1, r2 used for temporaries
- follows algorithm of our earlier data paths.

```
int n, ans;
r1 = 1;
r2 = n;
while (r2 != 0) {
   r1 = r1 * r2;
   r2 = r2 - 1
}
ans = r1;
```

Beta code, in assembly language:

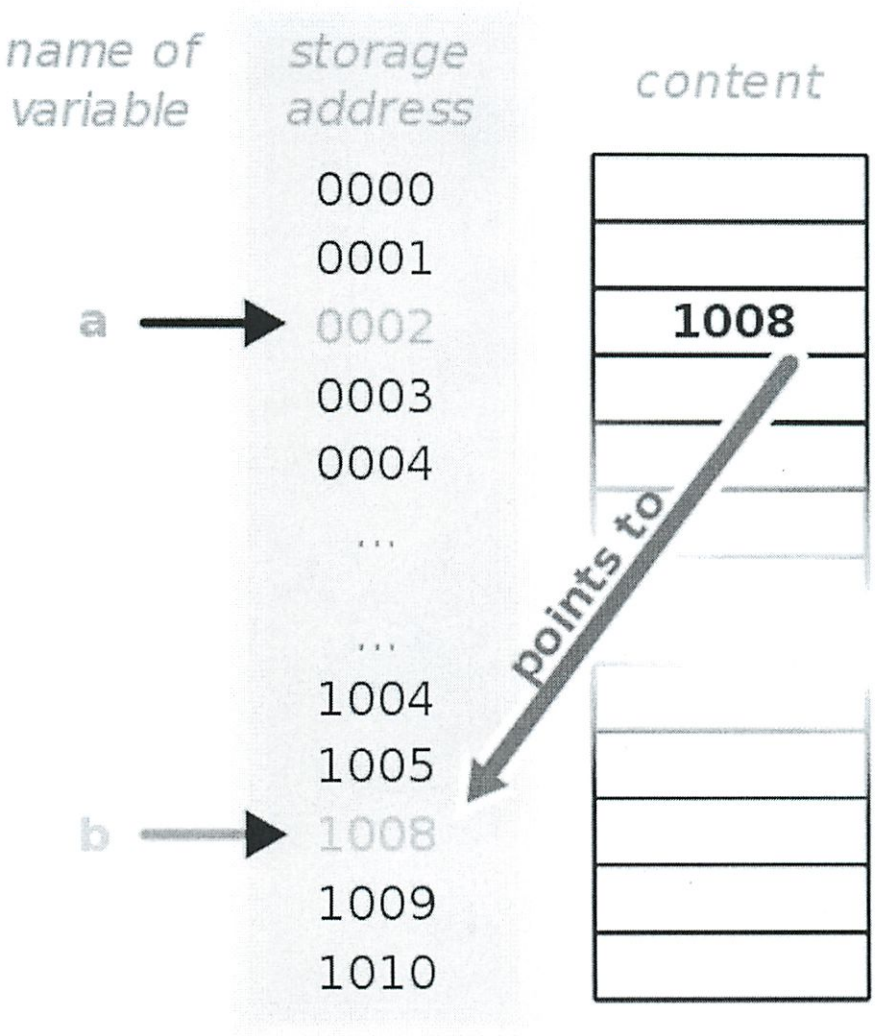
```
n:      long(123)
ans:    long(0)
        ...
        ADDC(r31, 1, r1)        | r1 = 1
        LD(n, r2)               | r2 = n
loop:   BEQ(r2, done, r31)      | while (r2 != 0)
        MUL(r1, r2, r1)         | r1 = r1 * r2
        SUBC(r2, 1, r2)         | r2 = r2 - 1
        BEQ(r31, loop, r31)     | Always branches!
done:   ST(r1, ans, r31)        | ans = r1
```


How implement in code?
With table of where to go next?
↳ like Turing machine
Or a more specific circuit implementation?

Summary

- Programmable data paths provide some algorithmic flexibility, just by changing control structure.
- Interesting control structure optimization questions – e.g., what operations can be done simultaneously?
- von Neumann model for general-purpose computation: need
  - support for sufficiently powerful operation repertoire
  - Expandable Memory
  - Interpreter for program stored in memory
- ISA design requires tradeoffs, usually based on benchmark results: art, engineering, evaluation & incremental optimizations
- Compilation strategy
  - runtime "discipline" for software implementation of a general class of computations
  - Typically enforced by compiler, run-time library, operating system. We'll see more of these!

Pointer



points to another memory area
take the place of registers in assemby largage
Used in trees and ~~[illegible]~~ look up tables
hold entry points for sub ratines

Reseach

10/18

WP: Pointer – see other sheet

Assembly language – representation of machine code
for a specific processor
assembler converts to machine instructions
– generally 1 to 1

RISC – reduced instruction set
– earlier processors good for manual assembly lang programming

Processor Register – small amt of storage in CPU
addressed differently than main memory
faster

So load data from main memory → do operation →
into register
save back to main memory
Save frequently accessed items in registers

Locality of reference – same value or storage repeatedly accessed
Canidate for caching, prefetching

②

<u>Stack</u> - the basic LIFO structure

But for hardware...

(see paper)

Basic architecture of a stack

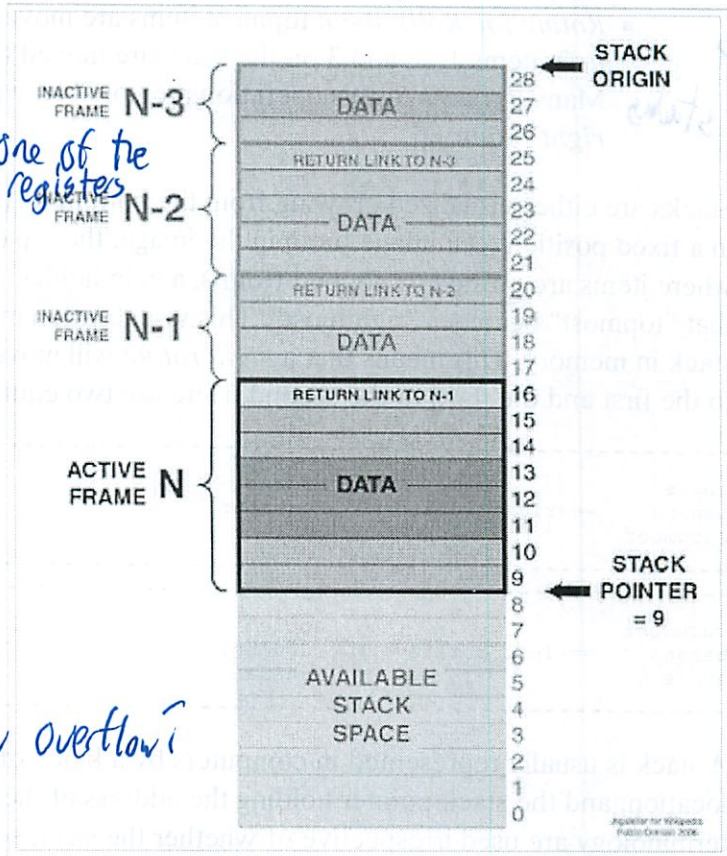
A typical stack is an area of computer memory with a fixed origin and a variable size. Initially the size of the stack is zero. A *stack pointer*, usually in the form of a hardware register, points to the most recently referenced location on the stack; when the stack has a size of zero, the stack pointer points to the origin of the stack.

one of the registers

The two operations applicable to all stacks are:

- a *push* operation, in which a data item is placed at the location pointed to by the stack pointer, and the address in the stack pointer is adjusted by the size of the data item;
- a *pop* or *pull* operation: a data item at the current location pointed to by the stack pointer is removed, and the stack pointer is adjusted by the size of the data item.

So this is how you overflow!



A typical stack, storing local data and call information for nested procedure calls (not necessarily nested procedures!). This stack grows downward from its origin. The stack pointer points to the current topmost datum on the stack. A push operation decrements the pointer and copies the data to the stack; a pop operation copies data from the stack and then increments the pointer. Each procedure called in the program stores procedure return information (in yellow) and local data (in other colors) by pushing them onto the stack. This type of stack implementation is extremely common, but it is vulnerable to buffer overflow attacks (see the text).

There are many variations on the basic principle of stack operations. Every stack has a fixed location in memory at which it begins. As data items are added to the stack, the stack pointer is displaced to indicate the current extent of the stack, which expands away from the origin.

Stack pointers may point to the origin of a stack or to a limited range of addresses either above or below the origin (depending on the direction in which the stack grows); however, the stack pointer cannot cross the origin of the stack. In other words, if the origin of the stack is at address 1000 and the stack grows downwards (towards addresses 999, 998, and so on), the stack pointer must never be incremented beyond 1000 (to 1001, 1002, etc.). If a pop operation on the stack causes the stack pointer to move past the origin of the stack, a *stack underflow* occurs. If a push operation causes the stack pointer to increment or decrement beyond the maximum extent of the stack, a *stack overflow* occurs.

how you define convention

Some environments that rely heavily on stacks may provide additional operations, for example:

- *Dup*(licate): the top item is popped, and then pushed again (twice), so that an additional copy of the former top item is now on top, with the original below it.
- *Peek*: the topmost item is inspected (or returned), but the stack pointer is not changed, and the stack size does not change (meaning that the item remains on the stack). This is also called **top**

So common fns for stacks

operation in many articles.

- *Swap* or *exchange*: the two topmost items on the stack exchange places.
- *Rotate (or Roll)*: the *n* topmost items are moved on the stack in a rotating fashion. For example, if *n*=3, items 1, 2, and 3 on the stack are moved to positions 2, 3, and 1 on the stack, respectively. Many variants of this operation are possible, with the most common being called *left rotate* and *right rotate*.

Stacks are either visualized growing from the bottom up (like real-world stacks), or, with the top of the stack in a fixed position (see image [note in the image, the top (28) is the stack 'bottom', since the stack 'top' is where items are pushed or popped from]), a coin holder, a Pez dispenser, or growing from left to right, so that "topmost" becomes "rightmost". This visualization may be independent of the actual structure of the stack in memory. This means that a *right rotate* will move the first element to the third position, the second to the first and the third to the second. Here are two equivalent visualizations of this process:

```
apple                          banana
banana     ===right rotate==>  cucumber
cucumber                       apple
```

```
cucumber                       apple
banana     ===left rotate==>   cucumber
apple                          banana
```

A stack is usually represented in computers by a block of memory cells, with the "bottom" at a fixed location, and the stack pointer holding the address of the current "top" cell in the stack. The top and bottom terminology are used irrespective of whether the stack actually grows towards lower memory addresses or towards higher memory addresses. (so this is a series of memory addresses?)

Pushing an item on to the stack adjusts the stack pointer by the size of the item (either decrementing or incrementing, depending on the direction in which the stack grows in memory), pointing it to the next cell, and copies the new top item to the stack area. Depending again on the exact implementation, at the end of a push operation, the stack pointer may point to the next unused location in the stack, or it may point to the topmost item in the stack. If the stack points to the current topmost item, the stack pointer will be updated before a new item is pushed onto the stack; if it points to the next available location in the stack, it will be updated *after* the new item is pushed onto the stack.

Popping the stack is simply the inverse of pushing. The topmost item in the stack is removed and the stack pointer is updated, in the opposite order of that used in the push operation.

Hardware support

Stack in main memory

Most CPUs have registers that can be used as stack pointers. Processor families like the x86, Z80, 6502, and many others have special instructions that implicitly use a dedicated (hardware) stack pointer to conserve opcode space. Some processors, like the PDP-11 and the 68000, also have special addressing modes for implementation of stacks, typically with a semi-dedicated stack pointer as well (such as A7 in the 68000). However, in most processors, several different registers may be used as additional stack pointers as needed (whether updated via addressing modes or via add/sub instructions).

Stack in registers or dedicated memory

The x87 floating point architecture is an example of a set of registers organised as a stack where direct access to individual registers (relative the current top) is also possible. As with stack-based machines in general, having the top-of-stack as an implicit argument allows for a small machine code footprint with a good usage of bus bandwidth and code caches, but it also prevents some types of optimizations possible on processors permitting random access to the register file for all (two or three) operands. A stack structure also makes superscalar implementations with register renaming (for speculative execution) somewhat more complex to implement, although it is still feasible, as exemplified by modern x87 implementations.

Sun SPARC, AMD Am29000, and Intel i960 are all examples of architectures using register windows within a register-stack as another strategy to avoid the use of slow main memory for function arguments and return values.

There are also a number of small microprocessors that implements a stack directly in hardware and some microcontrollers have a fixed-depth stack that is not directly accessible. Examples are the PIC microcontrollers, the Computer Cowboys MuP21, the Harris RTX line, and the Novix NC4016. Many stack-based microprocessors were used to implement the programming language Forth at the microcode level. Stacks were also used as a basis of a number of mainframes and mini computers. Such machines were called stack machines, the most famous being the Burroughs B5000.

Applications

Converting a decimal number into a binary number

The logic for transforming a decimal number into a binary number is as follows:

```
* Read a number
* Iteration (while number is greater than zero)
          1. Find out the remainder after dividing the number by 2
          2. Print the remainder
          3. Divide the number by 2
* End the iteration
```

However, there is a problem with this logic. Suppose the number, whose binary form we want to find is 23. Using this logic, we get the result as 11101, instead of getting 10111.

To solve this problem, we use a stack. We make use of the *LIFO* property of the stack. Initially we *push* the binary digit formed into the stack, instead of printing it directly. After the entire digit has been converted into the binary form, we *pop* one digit at a time from the stack and print it. Therefore we get the decimal number is converted into its proper binary form.

Algorithm:

```
1. Create a stack
2. Enter a decimal number, which has to be converted into its equivalent binary form.
3. iteration1 (while number > 0)
      3.1 digit = number % 2
      3.2 Push digit into the stack
      3.3 If the stack is full
           3.3.1 Print an error
```

(3)

Buffer overflow - write data over allocated area

Buffer - region of physical memory to temp hold stuff

Main memory - memory accessible to CPU
RAM
Harddrive is 2ndary storage

Word - ~~a~~ fixed group of bits, like 32 bits

6.004 Recitation

10/14

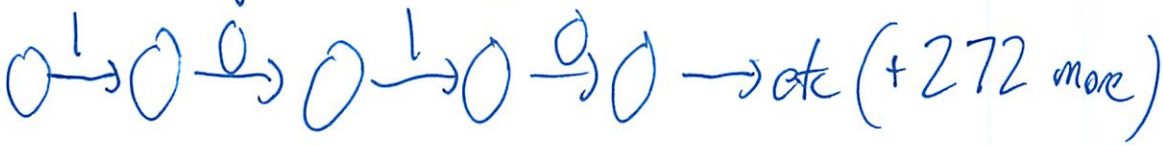
Quizzes back at end

First: Turing and computability

Later: Programmability

Can you calculate stuff using an FSM?

- is it bounded?
- like counting matching parentheses - unbounded, no
- last $\underbrace{277}_{\text{bounded, yes}}$ digits are alternating 0s + 1s
- FSMs are good at patterns



- More 0s than 1s - unbounded, no
- Divisable by 3?
  - can subtract 3 each time
    - number can be arbitrally large - so bad
  - is also an incremental, finite technique
    - so yes

- odd #1s, even #0s
  - think about how you would do it
  - 4 states
  - so yes

EE EO

OE OO

Turing Machines + Computability

- function <u>computable</u> iff there is a TM that takes x on input tape and outputs f(x)

  $TM_f[x] \rightsquigarrow f(x)$

- another def <u>computability</u>: there exists a finite program for computing answer
- might not be easy to know program
  - but could convince yourself that 1 exists

Fn HaltsBefore(k, j, s) = 1

└ $TM_k[j]$ halts before s steps

Fn Halts(k, j) ⊗ Not computable

(3)

But how about Halts Before?

Remember we have a universal TM
└ can emulate any TM

Yes. We can just run it S steps and see if it stops

fn H12345(x)

$TM_{12345}[12345]$ < 1 halts, 0 not

note! it does not pay attention to input X

Is computable - since just small program that returns value

But how to find answer in first case? - can't

You can't really build machine

But machine could exist

(confused...)

fn Dow(x)

{ Closing DOW year 2000+x $0 < x < 100$
{ 0 otherwise

(4)

You could do it in year 2200

└ So it is computability

But again, could not build such a machine

fn HZero(k)

if $TM_k[0]$ halts ?

↑ input is 0

- would need ∞ table

└ can't do w/ Turing machine

If this was computable, would it allow me to build a machine that solves the halting problem?

So this must not be computable

fn g(x, y) = i that writes y on the tape

then runs x TM on what it just wrote

$TM_x[y]$

Halts(k, j) = HZero(g(k, j))

(5)

These qu are on quiz
- Profs argue about the answers
  And then put it on the quiz anyway

You've shown us the argument to give it and it gives back if there is an answer or not
But how do you build this argument?
How do you build problem w/ blank tape?
- this last problem
  - I'm confused

Why are we interested in Computability?
- get insights in to how to solve certain problems
- ask if problem is hard or easy
- like cryptography
  - based on math proof that certain stuff is hard

6

Suppose given — Busy Beaver problem

- 2 state TM
  - + HALT
- zero tape

How many 1s can I write?

if no + halt; ∞

if halts; 4

Lots of talk about it online

- esp for n machines
- is uncomputable

If restricted to (0,1) on tape

can write 4 1s

Its like a big cross product

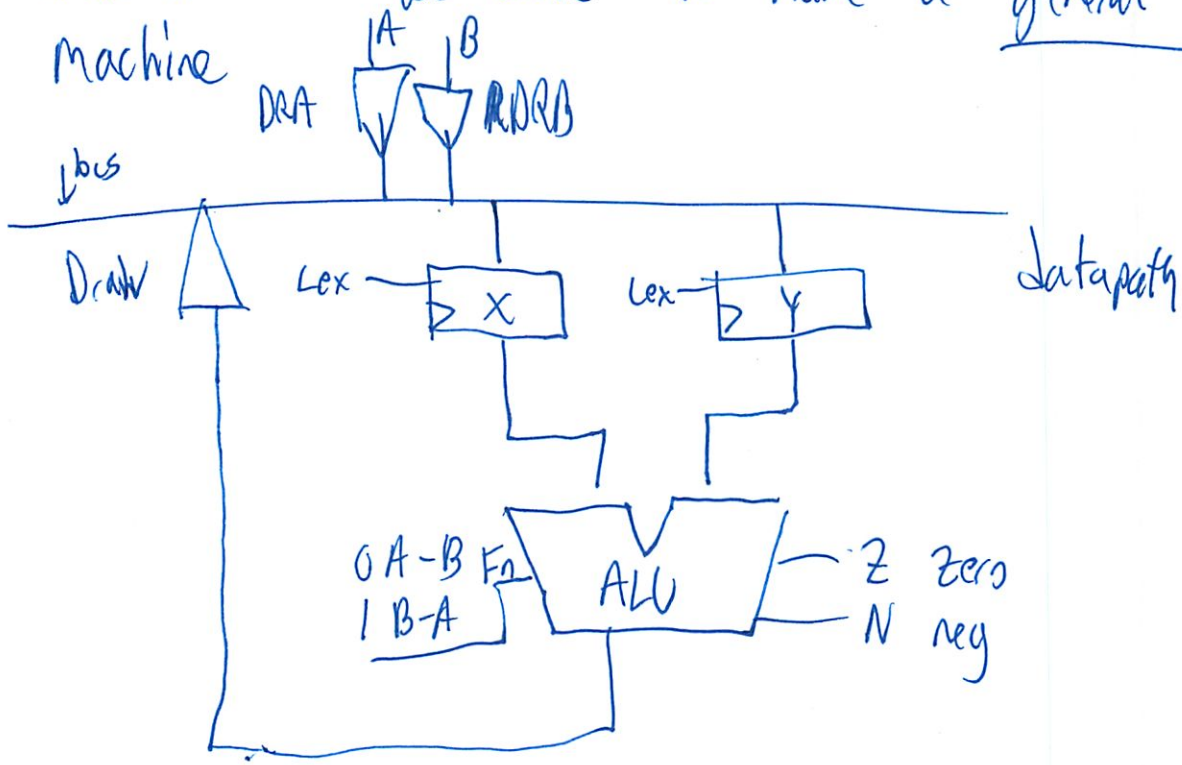
State × Symbols

2 × 2 = 4

How long can a TM run and then halt?

Actual Digital Hardware

- earliest computers were fixed purpose
- Wanted to be able to have a general purpose machine



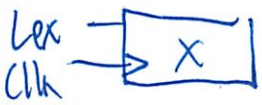
Ctrl	input
Dra	state
Drb	Z
Draw	N
Lex	
Ley	
Fn	

Want GCD

This is like a player piano

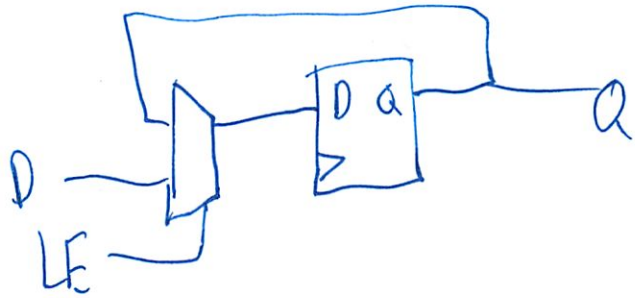
- punch a hole where you want a key press

Then search for function

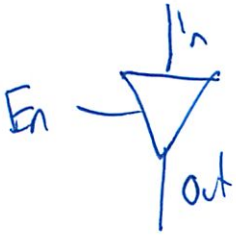


Load enabled register

- determines on next clk cycle if register loads
- internally it always loads
- but multiplexer says load new value or old value



Buffer



Can turn on or off

(9)

Must always have 1 thing driving the bus

Write GCD(a, b) algorithm

```
While (a ≠ b) {
    if (a > b)  a = a - b
    else        b = b - a
}
a, b = GCD  ← when done
```

State diagram that goes through steps

First need $a \to x$

$b \to y$

Can do both at once?

No! only 1 wire

Then ask does $a = b$?

if $z = 1$ we are done

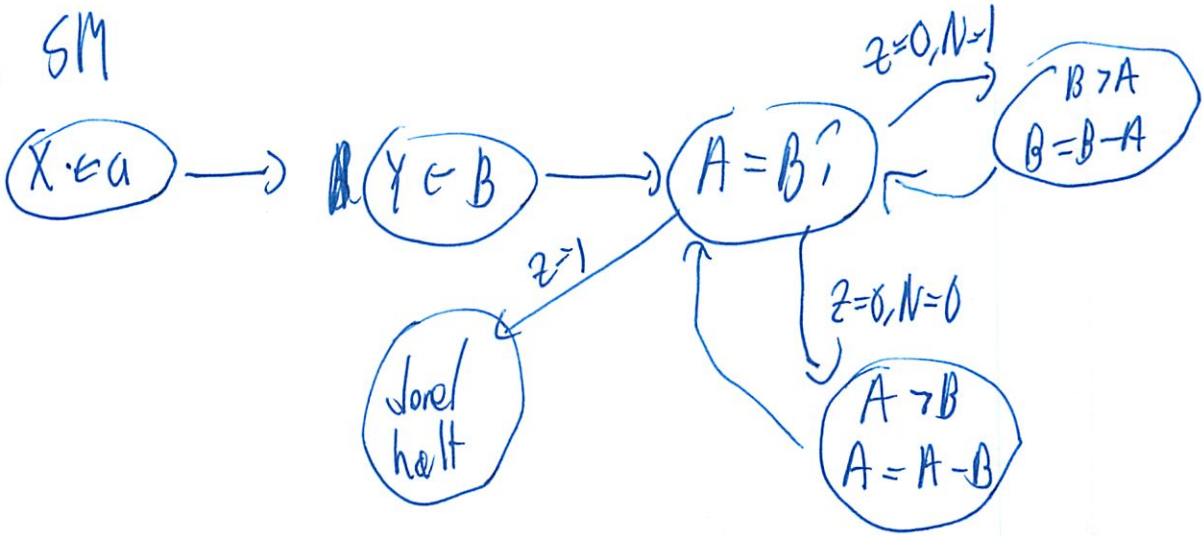
if $z = 0, N = 1$ $b > a$ so $b = b - a$

if $z = 0, N = 0$ a is $> b$

so $a = a - b$

(10)

Build SM



Then build truth table

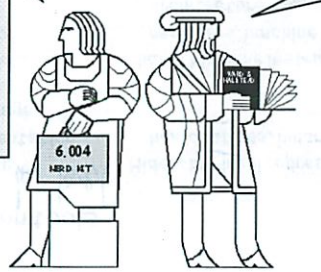
See rest online

10/20

Machine Language, Assemblers, and Compilers

Long, long, time ago, I can still remember
how mnemonics used to make me smile...
And I knew that with just the opcode names
that I could play those BSim games
and maybe hack some macros for a while.
But 6.004 gave me shivers
with every lecture they delivered.
Bad news at the door step,
I couldn't read one more spec.
I can't remember if I tried
to get Factorial optimized,
But something touched my nerdish pride
the day my Beta died.
And I was singing...

When I find my code in tons of trouble,
Friends and colleagues come to me,
Speaking words of wisdom:
"Write in C."



Software tools for Beta

References:
β Documentation
BSIM reference
Notes on C Language

Lab 4 due TODAY!

β Machine Language: 32-bit instructions

OPCODE	r_c	r_a	r_b	*unused*

operations

arithmetic: ADD, SUB, MUL, DIV
compare: CMPEQ, CMPLT, CMPLE
boolean: AND, OR, XOR
shift: SHL, SHR, SRA

Ra and Rb are the operands,
Rc is the destination.
R31 reads as 0, unchanged by writes

OPCODE	r_c	r_a	16-bit signed constant

arithmetic: ADDC, SUBC, MULC, DIVC
compare: CMPEQC, CMPLTC, CMPLEC
boolean: ANDC, ORC, XORC
shift: SHLC, SHRC, SRAC
branch: BNE/BT, BEQ/BF (const = word displacement from PC_{NEXT})
jump: JMP (const not used)
memory access: LD, ST (const = byte offset from Reg[ra])

Two's complement 16-bit constant for numbers from −32768 to 32767; sign-extended to 32 bits before use.

How can we improve the programmability of the Beta?

don't want to write 32 bit strings!

Encoding Binary Instructions

32-bit (4-byte) ADD instruction:

100000	00100	00010	00011	0000000000
OpCode	Rc	Ra	Rb	(unused)

Means, to BETA, Reg[4] = Reg[2] + Reg[3]

But, most of us would prefer to write

`ADD(R2, R3, R4)` *(ASSEMBLER)*

have PC translate to bits

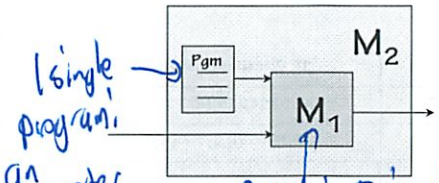
or, better yet,

`a = b+c;` no op-code *(High Level Language)*

we'll do it on quiz 3

Software Approaches: INTERPRETATION, COMPILATION

Interpretation



1 single program, an interpreter

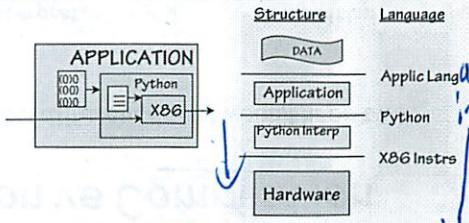
proved is Turing universal

Turing's model of *Interpretation:*

- Start with some hard-to-program *universal* machine, say M_1
- Write a single program for M_1 which mimics the behavior of some easier machine, say M_2
- Result: a "virtual" M_2

"Layers" of interpretation:

- Often we use several layers of interpretation to achieve desired behavior, eg:
- X86 (Pentium), running
  - Python, running
    - Application, interpreting
      - Data.



app could interpret

higher level data

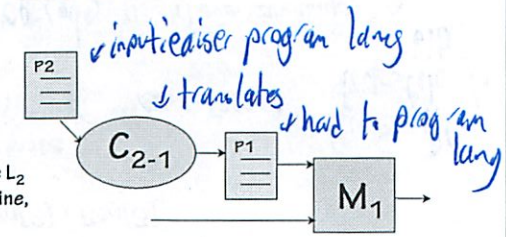
10/20

in real life, many more

Compilation

Model of *Compilation:*

- Given some hard-to-program machine, say M_1...
- Find some easier-to-program language L_2 (perhaps for a more complicated machine, M_2); write programs in that language
- Build a translator (compiler) that translates programs from M_2's language to M_1's language. May run on M_1, M_2, or some other machine.



↓ input: easier program lang
↓ translates
↓ hard to program lang

Interpretation & Compilation: two tools for improving programmability ...

- *Both* allow changes in the programming model
- *Both* afford programming applications in platform (e.g., processor) independent languages
- *Both* are widely used in modern computer systems!

Interpretation vs Compilation

There are some characteristic differences between these two powerful tools...

	Interpretation	Compilation
How it treats input "x+2"	computes x+2	generates a program that computes x+2
When it happens	During execution	Before execution
What it complicates/slows	Program Execution	Program Development
Decisions made at	Run Time	Compile Time

Major design choice we'll see repeatedly: do it at Compile time or at Run time?

When do we make the decision?

Software: Abstraction Strategy

Initial steps: compilation tools

Assembler (UASM): symbolic representation of machine language	Hides: bit-level representations, hex locations, binary values
Compiler (C): symbolic representation of algorithm	Hides: Machine instructions, registers, machine architecture

low level

builds the 32 bit word

lowest of higher level lang

Subsequent steps: interpretive tools

Operating system	Hides: Resource (memory, CPU, I/O) limitations and details
Apps (e.g., Browser)	Hides: Network; location; local parameters

later on 6.004

Abstraction step 1:

A Program for Writing Programs

UASM - the 6.004 (Micro) Assembly Language

STREAM of Bytes to be loaded into memory

Symbolic SOURCE text file

UASM Translator program (built into BSIM)

Binary Machine Language

simple rules

initial config & memory

UASM:

1. A Symbolic LANGUAGE for representing strings of bits
2. A PROGRAM ("assembler" = primitive compiler) for translating UASM source to binary.

UASM Source Language

32 bit string is like series of 8 bit bytes

A UASM SOURCE FILE contains, in symbolic text, values of successive bytes to be loaded into memory... e.g. in

consecutive locations on mem

```
37    -3    255      decimal (default);
0b100101             binary (note the "0b" prefix);
0x25                 hexadecimal (note the "0x" prefix);
```

2s complement

Values can also be expressions; eg, the source file

```
37+0b10-0x10    24-0x1    4*0b110-1    0xF7&0x1F
```

← 4 expressions

each the same

generates 4 bytes of binary output, each with the value 23!

Symbolic Gestures

A "bar" denotes the beginning of a comment... The remainder of the line is ignored

We can also define SYMBOLS for use in source programs:

```
x = 0x1000          | A variable location
y = 0x1004          | Another variable

| Symbolic names for registers:
R0 = 0
R1 = 1
 ...
R31 = 31
```

Special variable "." (period) means next byte address to be filled:

"dot" ← set next 4 + increment

```
set →  . = 0x100          | Assemble into 100
         1  2  3  4
read →  five = .          | Symbol "five" is 0x104
         5  6  7  8
       . = .+16           | Skip 16 bytes
         9 10 11 12
```

0 t locations

Labels (Symbols for Addresses)

LABELS are symbols that represent memory addresses. They can be set with the following special syntax:

"tags" **x:** is an abbreviation for "**x = .**"

An Example--

```
---- MAIN MEMORY ----        . = 0x1000
  1000: 09 04 01 00          sqrs: 0 1 4 9
  1004: 31 24 19 10                16 25 36 49
  1008: 79 64 51 40                64 81 100 121
  100c: E1 C4 A9 90                144 169 196 225
  1010: 00 00 00 10          slen: LONG(. - sqrs)
        3  2  1  0
```

start here

call sqrs to be .

load these data

slen →

store in slen the lenght of table in 32 bit integer

sqrs is 0x1000 ← at the beginning

Mighty Macroinstructions

it doesn't even know add to start

long extension mechanism

Macros are parameterized abbreviations, or shorthand

```
| Macro to generate 4 consecutive bytes:
.macro consec(n)  n  n+1  n+2  n+3

| Invocation of above macro:
consec(37)
```

declare template

Has same effect as:

```
37    38    39    40
```

Simple, but not useful macro

Here are macros for breaking multi-byte data types into byte-sized chunks

```
| Assemble into bytes, little-endian:
.macro WORD(x)  x%256  (x/256)%256
.macro LONG(x)  WORD(x)  WORD(x >> 16)

. = 0x100
LONG(0xdeadbeef)
```

builds 16 bit

Boy, that's hard to read. Maybe, those big-endian types do have a point.

Has same effect as:

```
        0xef   0xbe   0xad   0xde
Mem:    0x100  0x101  0x102  0x103
```

byte order - low order bit - little endian
or big order bit - big endian

Very expensive, since it is split

beta is little endian

animation

-32768 = 1000000000000000

Assembly of Instructions

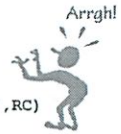
don't spend much time on

OPCODE	RC	RA	RB	UNUSED
110000	00000	01111	1000000000000000	

```
| Assemble Beta op instructions
.macro betaop(OP,RA,RB,RC) {
    .align 4
    LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+((RB%32)<<11))
}

| Assemble Beta opc instructions
.macro betaopc(OP,RA,CC,RC) {
    .align 4
    LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+(CC % 0x10000))
}

| Assemble Beta branch instructions
.macro betabr(OP,RA,RC,LABEL)      betaopc(OP,RA,((LABEL-(.+4))>>2),RC)
```

"**.align 4**" ensures instructions will begin on word boundary (i.e., address = 0 mod 4)

generate 32 bit word




```
For Example:
        ADDC(R15, -32768, R0) --> betaopc(0x30,15,-32768,0)
```


Finally, Beta Instructions

On for each Beta instruction

```
| BETA Instructions:
.macro ADD(RA,RB,RC)    betaop(0x20,RA,RB,RC)
.macro ADDC(RA,C,RC)    betaopc(0x30,RA,C,RC)
.macro AND(RA,RB,RC)    betaop(0x28,RA,RB,RC)
.macro ANDC(RA,C,RC)    betaopc(0x38,RA,C,RC)
.macro MUL(RA,RB,RC)    betaop(0x22,RA,RB,RC)
.macro MULC(RA,C,RC)    betaopc(0x32,RA,C,RC)
   :
   .
.macro LD(RA,CC,RC)     betaopc(0x18,RA,CC,RC)
.macro LD(CC,RC)        betaopc(0x18,R31,CC,RC)
.macro ST(RC,CC,RA)     betaopc(0x19,RA,CC,RC)
.macro ST(RC,CC)        betaopc(0x19,R31,CC,RC)
   :
   .
.macro BEQ(RA,LABEL,RC) betabr(0x1C,RA,RC,LABEL)
.macro BEQ(RA,LABEL)    betabr(0x1C,RA,r31,LABEL)
.macro BNE(RA,LABEL,RC) betabr(0x1D,RA,RC,LABEL)
.macro BNE(RA,LABEL)    betabr(0x1D,RA,r31,LABEL)
```

Convenience macros so we don't have to specify R31...

(from beta.uasm)

Example Assembly

straight forward

ADDC(R3,1234,R17)

⇩ *expand ADDC macro with RA=R3, C=1234, RC=R17*

betaopc(0x30,R3,1234,R17)

⇩ *expand betaopc macro with OP=0x30, RA=R3, CC=1234, RC=R17*

.align 4
LONG((0x30<<26)+((R17%32)<<21)+((R3%32)<<16)+(1234 % 0x10000))

⇩ *expand LONG macro with X=0xC22304D2*

WORD(0xC22304D2) WORD(0xC22304D2 >> 16)

⇩ *expand first WORD macro with X=0xC22304D2*

0xC22304D2%256 (0xC22304D2/256)%256 WORD(0xC223)

⇩ *evaluate expressions, expand second WORD macro with X=0xC223*

0xD2 0x04 0xC223%256 (0xC223/256)%256

⇩ *evaluate expressions*

0xD2 0x04 0x23 0xC2

R31 is always 0

Don't have it? Fake it!

not directly defined w/ op codes

Convenience macros can be used to extend our assembly language:

```
.macro MOVE(RA,RC)        ADD(RA,R31,RC)     | Reg[RC] <- Reg[RA]
.macro CMOVE(CC,RC)       ADDC(R31,C,RC)     | Reg[RC] <- C

.macro COM(RA,RC)         XORC(RA,-1,RC)     | Reg[RC] <- ~Reg[RA]
.macro NEG(RB,RC)         SUB(R31,RB,RC)     | Reg[RC] <- -Reg[RB]
.macro NOP()              ADD(R31,R31,R31)   | do nothing

.macro BR(LABEL)          BEQ(R31,LABEL)     | always branch
.macro BR(LABEL,RC)       BEQ(R31,LABEL,RC)  | always branch
.macro CALL(LABEL)        BEQ(R31,LABEL,LP)  | call subroutine
.macro BF(RA,LABEL,RC)    BEQ(RA,LABEL,RC)   | 0 is false
.macro BF(RA,LABEL)       BEQ(RA,LABEL)
.macro BT(RA,LABEL,RC)    BNE(RA,LABEL,RC)   | 1 is true
.macro BT(RA,LABEL)       BNE(RA,LABEL)

| Multi-instruction sequences
.macro PUSH(RA)           ADDC(SP,4,SP)   ST(RA,-4,SP)
.macro POP(RA)            LD(SP,-4,RA)    ADDC(SP,-4,SP)
```

convience macros

1 = True
0 = False

(from beta.uasm)

Abstraction step 2:

High-level Languages

Most algorithms are naturally expressed at a high level. Consider the following algorithm:

```
struct Employee
 { char *Name;  /* Employee's name. */
   long Salary; /* Employee's salary. */
   long Points;}/* Brownie points. */

/* Annual raise program. */
Raise(struct Employee P[100])
 { int i = 0;
   while (i < 100)
    { struct Employee *e = &P[i];
      e->Salary =
           e->Salary + 100 + e->Points;
      e->Points = 0;  /* Start over! */
      i = i+1;
    }
 }
```

We've used (and will continue to use throughout 6.004) C, a "mature" and common systems programming language. Modern popular alternatives include C++, Java, Python, and many others.

relatively machine ind.

Why use these, not assembler?

- readable
- concise
- unambiguous
- portable (algorithms frequently outlast their HW platforms)
- Reliable (type checking, etc)

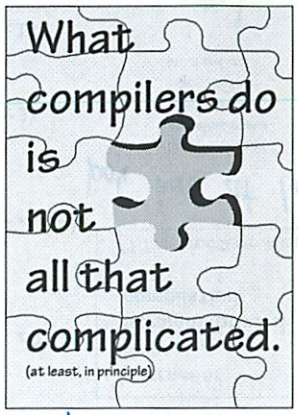
Reference: C handout (6.004 web site)

easy, but a few tricks

How Compilers Work

Contemporary compilers go far beyond the macro-expansion technology of UASM. They

- Perform sophisticated analyses of the source code
- Invoke arbitrary algorithms to generate efficient object code for the target machine
- Apply "optimizations" at both source and object-code levels to improve run-time efficiency.



Compilation to unoptimized code is pretty straightforward... following is a brief glimpse.

easy if don't optimize

Compiling Expressions

C code:

```
int x, y;
y = (x-3)*(y+123456)
```

translate

Beta assembly code:

```
x:      LONG(0)
y:      LONG(0)
c:      LONG(123456)
...
        LD(x, r1)
        SUBC(r1,3,r1)
        LD(y, r2)
        LD(C, r3)
        ADD(r2,r3,r2)
        MUL(r2,r1,r1)
        ST(r1,y)
```

load — SUBC — store

compiler sees

not 16 bit so must do

x:	
y:	
c:	123456

Rules

- VARIABLES are assigned memory locations and accessed via LD or ST
- OPERATORS translate to ALU instructions
- SMALL CONSTANTS translate to "literal-mode" ALU instructions
- LARGE CONSTANTS translate to initialized variables

2 classes that are important

Data Structures: Arrays

The C source code

```
int Hist[100];
...
Hist[score] += 1;
```

might translate to:

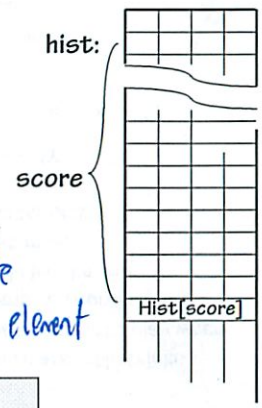
```
hist:   .=.+4*100 | Leave room for 100 ints
...
<score in r1>
MULC(r1,4,r2)     | index -> byte offset
LD(r2,hist,r0)    | hist[score]
ADDC(r0,1,r0)     | increment
ST(r0,hist,r2)    | hist[score]
```

refer. to by seq #

offset

must compute address of element

Memory:



Address:
CONSTANT base address +
VARIABLE offset computed from index

Records/

Data Structures: Structs

```
struct Point
 { int x, y;
 } P1, P2, *p;
...
P1.x = 157;
...
p = &P1;
p->y = 157;
```

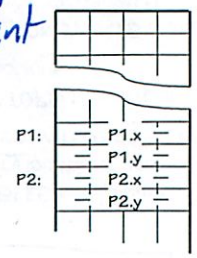
← define 2 variables

↑ 3rd variable – pointer to Point

← address of

↓ component of what p sets to

Memory:



might translate to:

```
P1: .=.+8
P2: .=.+8
x=0                 | Offset for x component
y=4                 | Offset for y component
...
ADDC(r31,157,r0)      | r0 <- 157
ST(r0,P1+x)           | P1.x = 157
...
<p in r3>
ST(r0,y,r3)           | p->y = 157;
```

↑ offset

Set at compile time

Address:
VARIABLE base address +
CONSTANT component offset

Conditionals

C code:

```
if (expr)
 {
    STUFF
 }
```

Beta assembly: Branch

```
(compile expr into rx)
BF(rx, Lendif)
(compile STUFF)
Lendif:
```

C code:

```
if (expr)
 {
    STUFF1
 }
else
 {
    STUFF2
 }
```

Beta assembly:

```
(compile expr into rx)
BF(rx, Lelse)
(compile STUFF1)
BR(Lendif)
Lelse:
(compile STUFF2)
Lendif:
```

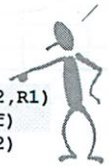
There are little tricks that come into play when compiling conditional code blocks. For instance, the statement:

```
if (y > 32)
 {
    x = x + 1;
 }
```

there's no > instruction!

compiles to:

```
LD(y,R1)
CMPLEC(R1,32,R1)
BT(R1,Lendif)
ADDC(R2,1,R2)
Lendif:
```



not spending on tricks

most time executing is in loops

Loops

are particularly imp

Move the test to the end of the loop and branch there the first time thru... saves a branch

C code:

```
while (expr)
 {
    STUFF
 }
```

Beta assembly:

```
Lwhile:
    (compile expr into rx)
    BF(rx,Lendwhile)
    (compile STUFF)
    BR(Lwhile)
Lendwhile:
```

Alternate Beta assembly:

```
    BR(Ltest)
Lwhile:
    (compile STUFF)
Ltest:
    (compile expr into rx)
    BT(rx,Lwhile)
Lendwhile:
```

Compilers spend a lot of time optimizing in and around loops.
- moving all possible computations outside of loops
- "*unrolling*" loops to reduce branching overhead
- simplifying expressions that depend on "loop variables"

Our Favorite Program

```
int n = 20, r;      n: LONG(20)
                    r: LONG(0)
                    start:
r = 1;                 ADDC(r31, 1, r0)
                       ST(r0, r)
                    loop:
                       LD(n, r1)
while (n > 0)          CMPLT(r31, r1, r2)
{                      BF(r2, done)
                       LD(r, r3)
                       LD(n,r1)
  r = r*n;             MUL(r1, r3, r3)
                       ST(r3, r)
                       LD(n,r1)
                       SUBC(r1, 1, r1)
  n = n-1;             ST(r1, n)
                       BR(loop)
}                   done:
```

Cleverness:
None...
straightforward compilation

(11 instructions in loop...)

look hard at loop to make it faster

Optimizations are what make ~~compilers complicated~~ interesting!

doing a lot of loading + storing

Optimizations

```
int n = 20, r;      n: LONG(20)
                    r: LONG(0)

r = 1;              start:
                       ADDC(r31, 1, r0)
                       ST(r0, r)
                       LD(n,r1)      | keep n in r1
                       LD(r,r3)      | keep r in r3

                    loop:
while (n > 0)          CMPLT(r31, r1, r2)
{                      BF(r2, done)
  r = r*n;             MUL(r1, r3, r3)
  n = n-1;             SUBC(r1, 1, r1)
}                      BR(loop)

                    done:
                       ST(r1,n)      | save final n
                       ST(r3,r)      | save final r
```

store when done

Cleverness:
We move LDs/STs out of loop!

(Still, 5 instructions in loop...)

Really Optimizing...

```
int n = 20, r;      n: LONG(20)
                    r: LONG(0)

                    start:
r = 1;                 LD(n,r1)          | keep n in r1
                       ADDC(r31,1,r3)    | keep r in r3
                       BEQ(r1, done)     | why?
                    loop:
while (n > 0)          MUL(r1, r3, r3)
{ r = r*n;             SUBC(r1, 1, r1)
  n = n-1;             BNE(r1,loop)
}                   done:
                       ST(r1,n)          | save final n
                       ST(r3,r)          | save final r
```

use branch instructions themselves to do loop

Cleverness:
We avoid overhead of conditional!

(Now 3 instructions in loop...)

but bug! - assume we will do it at least once

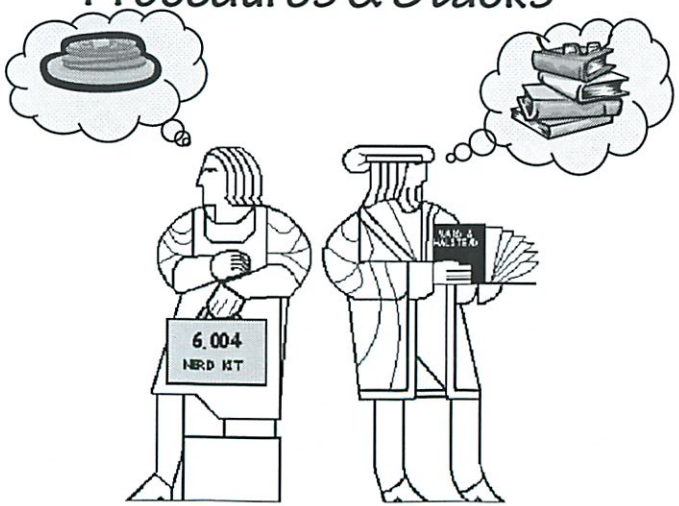
UNFORTUNATELY,
20! = 2,432,902,008,176,640,000 > 2^{61} *(overflows!)*
but 12! = 479,001,600 = 0x1c8cfc00

so need to check if n=0 1st

Compilers do this and more

Coming Attractions:

Procedures & Stacks

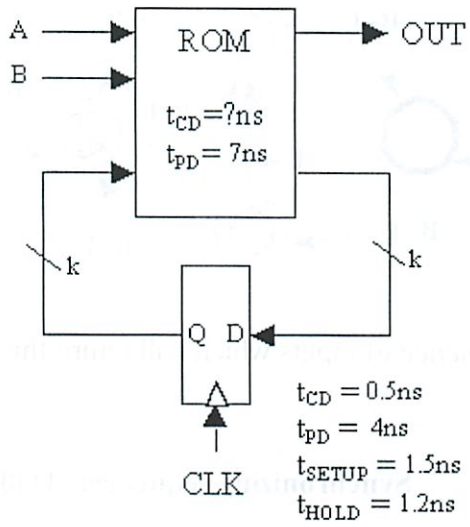


6.004 On-line: Questions for Lab 4

When you're done remember to save your work by clicking on the "Save" button at the bottom of the page. You can check if your answers are correct by clicking on the "Check" button.

When entering numeric values in the answer fields, you can use integers (1000), floating-point numbers (1000.0), scientific notation (1e3), or JSim numeric scale factors (1K).

Problem 1. A possible implementation of a finite state machine with two inputs and one output is shown below.



A. If the register is 5 bits wide (i.e., k = 5) what is the appropriate size of the ROM? Give the number of locations and the number of bits in each location.

Number of locations: 128 ✓

Number of bits in each location: 6 ✓

B. If the register is 5 bits wide what is the maximum number of states in an FSM implemented using this circuit?

Maximum number of states: 32 ✓

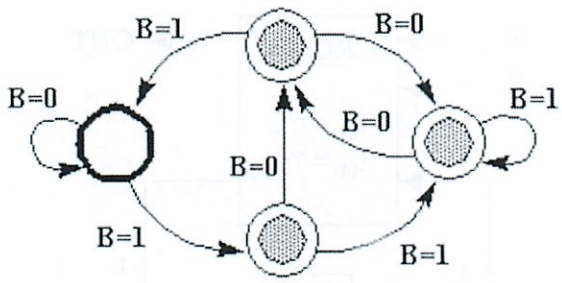
C. What is the smallest possible value for the ROM's contamination delay that still ensures the necessary timing specifications are met?

smallest possible value for $t_{CD\text{-}ROM}$ (in seconds): .7ns ✓

D. Assume that the ROM's t_{CD} = 3ns. What is the shortest possible clock period that still ensures that the necessary timing specifications are met?

smallest clock period (in seconds): 12.5ns

Problem 2. Shown below is a state transition diagram for an FSM, F, with a single binary input B. The FSM has a single output, a light which is on for the three states marked by a gray dot. The starting state is marked by the heavy circle.



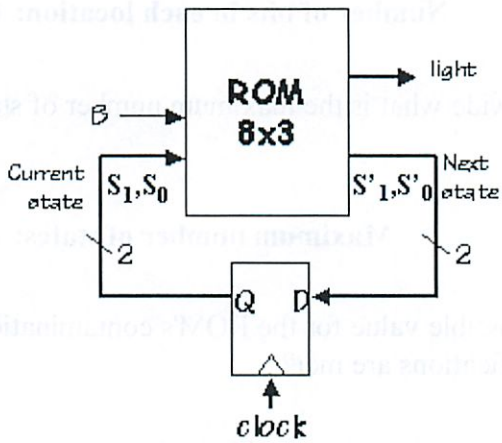
A. Is there a *synchronizing sequence* of inputs which will return this FSM from an unknown state to its starting state?

Synchronizing sequence: 11101 is such a sequence

B. Does this FSM have a pair of equivalent states that may be merged to yield a 3-state FSM?

Equivalent states: Yes; the lower and rightmost states are equivalent.

C. The following circuit is used to implement the above 4-state FSM:

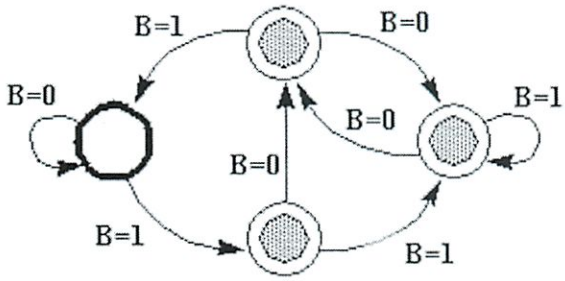


It is known that the starting state of the 4-state FSM corresponds to 00 on the state variable input, and the **light** output is 1 when the light is to be on. What is the value of the **light** output

when all three inputs to the ROM are zero?

Value of light output: 0 (light off)

D. Fill in the unspecified rows of the following truth table so that it implements the state transition diagram. You will need to enter some combination of three zeros or ones in each field. Other characters in the fields (e.g., spaces) will be ignored. Remember the starting state is 00.



S1	S0	B	S1'	S0'	light
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1
1	1	1	1	1	1

Check Save

source: on_line_questions.py, lab4questions.xdoc

6.004
Lab 4 Qu

10/20

1. Implementation of FSM

register = 5 bits

What should size of ROM be?

2 inputs

$2^{2} \cdot 5$ = # of gates

(#inputs; cbits; always 2)

Or 4 locations ⊗

5 bits / location ⊗

I don't get why not!

I have no clue!

Roms have 2^k signals for k inputs

And column for each – so I think my original ans is right!

(2)

b) Skip for now

Max # of state

↳ well either can be 0 or 1

2^5 (✓)

c) Smallest t_{CD} for ROM

? depends on each gate

t_{CD} is <u>shortest</u> way through

So 1 time step?

We never specifically did ROM timing

X 1.5ns → 6ns in 1.5 ns intervals not taken

Skip

d) What is shortest t_{clk}? $t_{CD} = 3$ ns

This was all last exam

I removed papers...

$t_{clk} \leq \sum t_{pd} + t_s$?

$7 + 1.5$ (X)

(3)

They give tokn should use that?

Skip till I get my notes

---

2. Is there a sync seq that always returns to start state from any starting state?

001?

No, does not work in starting state

1001

No

Oh they have suggestions

Last ~~one~~ digit must be 1

11101 ✓

b) Is there an equiv for 3 states

↳ To say yes would have to draw

How do you rigerously determine?

(4)

Are suggestions

Lower and right most

Yes 1 points to itself each time
0 points to same place

Have an 8 x 3 rom

$8 = 2^2$?
↑ where is this

$8 = 2^4$ (x)

So where is 8 from?

Should be

$2^{\#\text{inputs}} \times \#\text{outputs}$

↑ that is 2 ↑ this is 2

State is like 00
When $B, S_1, S_0 = 0$

So at off state, a 0 B=0 → (00) (0)

(5)

Fill in state transition diagram

└ Yeah I never rigorously practiced this
– did it right on exam

S_1	S_0	B
0	0	0

0	0	0

← redirects

But what is internal state – can't really see
Or can from other values on table

know 2 transitions

10 ← can guess

except 0 0 1

I said 1 0 1

Answer 1 00 ← is that correct

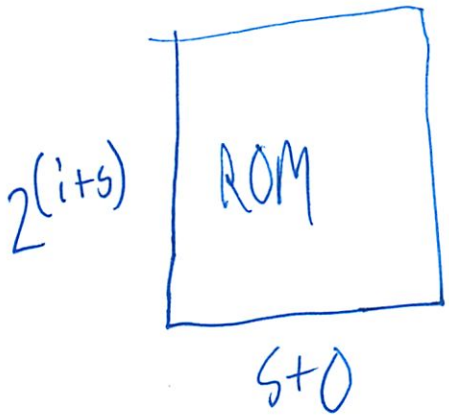
At 1 0, light is always on!

So done except timing qu

6.004

Ask chris

10/20



So size $= 2^{(i+s)} \cdot (s+o)$

i = inputs

s = state bits

o = outputs

10/20

6.004
Lab Qu More

Locations $= 2^{(i+s)}$

What are # states

output $= 5$

that is state

2^{2+5} ✓

bits each location

0 +5

1 +5 ✓

Max # states

$32 = 2^5$

Timing

↓

(2)

Timing for Roms

T_{CD} - Min of possible paths

Mux + CMOS + Inv

TA: Want so timing spec is met

Worse case behavior

Related to register

~~Min contamination~~

Use these rules

$t_{CD\ Reg} + t_{C\ Logic} > t_{Hold\ Reg2}$

$.5ns + \_\_\_\_ \geq 1.2ns$

$.7ns$ (✓)

Reg1 = Reg2 since lab

$t_{Clk} \geq t_{PD\ Reg1} + t_{PD\ Logic} + t_{setup\ Reg1}$

$4ns + 7ns + 1.5ns$

$12.5ns$ (✓)

(✓) Questions complete

Turing machine

From Wikipedia, the free encyclopedia

A **Turing machine** is a theoretical device that manipulates symbols on a strip of tape according to a table of rules. Despite its simplicity, a Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of a CPU inside a computer.

The "Turing" machine was described by Alan Turing in 1936,[1] who called it an "*a*(utomatic)-machine". The Turing machine is not intended as a practical computing technology, but rather as a thought experiment representing a computing machine. Turing machines help computer scientists understand the limits of mechanical computation.

Turing gave a succinct definition of the experiment in his 1948 essay, "Intelligent Machinery". Referring to his 1936 publication, Turing wrote that the Turing machine, here called a Logical Computing Machine, consisted of:

> ...an infinite memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed. At any moment there is one symbol in the machine; it is called the scanned symbol. The machine can alter the scanned symbol and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine. However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings.[2] (Turing 1948, p. 61)

programability

A Turing machine that is able to simulate any other Turing machine is called a universal Turing machine (**UTM**, or simply a **universal machine**). A more mathematically-oriented definition with a similar "universal" nature was introduced by Alonzo Church, whose work on lambda calculus intertwined with Turing's in a formal theory of computation known as the Church–Turing thesis. The thesis states that Turing machines indeed capture the informal notion of effective method in logic and mathematics, and provide a precise definition of an algorithm or 'mechanical procedure'.

Studying their abstract properties yields many insights into computer science and complexity theory.[3]

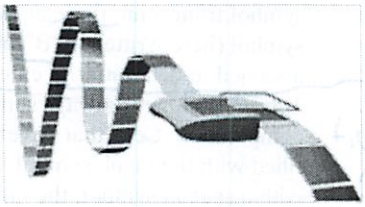
Turing machine(s)

Machina

- Universal Turing machine
- Alternating Turing machine
- Quantum Turing machine
- Read-only Turing machine
- Read-only right moving Turing Machines
- Probabilistic Turing machine
- Multi-track Turing machine
- Turing machine equivalents
- Turing machine examples
- Wolfram's 2-state 3-symbol..

Science

- Alan Turing
- Category:Turing machine



An artistic representation of a Turing machine (Rules table not represented)

Contents

Informal description

For visualizations of Turing machines, see Turing machine gallery.

The Turing machine mathematically models a machine that mechanically operates on a tape. On this tape are symbols which the machine can read and write, one at a time, using a tape head. Operation is fully determined by a finite set of elementary instructions such as "in state 42, if the symbol seen is 0, write a 1; if the symbol seen is 1, shift to the right, and change into state 17; in state 17, if the symbol seen is 0, write a 1 and change to state 6;" etc. In the original article ("On computable numbers, with an application to the Entscheidungsproblem", see also references below), Turing imagines not a mechanism, but a person whom he calls the "computer", who executes these deterministic mechanical rules slavishly (or as Turing puts it, "in a desultory manner").

state is internal

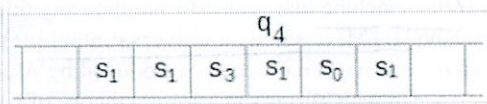
More precisely, a Turing machine consists of:

1. A **tape** which is divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains a special *blank* symbol (here written as 'B') and one or more other symbols. The tape is assumed to be arbitrarily extendable to the left and to the right, i.e., the Turing machine is always supplied with as much tape as it needs for its computation. Cells that have not been written to before are assumed to be filled with the blank symbol. In some models the tape has a left end marked with a special symbol; the tape extends or is indefinitely extensible to the right.

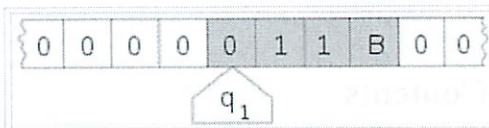
the ∞ part - doesn't wrap around or something

2. A **head** that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time. In some models the head moves and the tape is stationary.
3. A finite **table** (occasionally called an **action table** or **transition function**) of instructions (usually quintuples [5-tuples]: $q_i a_j \rightarrow q_{i1} a_{j1} d_k$, but sometimes 4-tuples) that, given the *state*(q_i) the machine is currently in *and* the *symbol*(a_j) it is reading on the tape (symbol currently under the head) tells the machine to do the following in sequence (for the 5-tuple models):
   - Either erase or write a symbol (instead of a_j, write a_{j1}), *and then*
   - Move the head (which is described by d_k and can have values: 'L' for one step left *or* 'R' for one step right *or* 'N' for staying in the same place), *and then*
   - Assume the same or a *new state* as prescribed (go to state q_{i1}).

   In the 4-tuple models, erase or write a symbol (a_{j1}) and move the head left or right (d_k) are specified as separate instructions. Specifically, the table tells the machine to (ia) erase or write a symbol *or* (ib) move the head left or right, *and then* (ii) assume the same or a new state as prescribed, but not both actions (ia) and (ib) in the same instruction. In some models, if there is no entry in the table for the current combination of symbol and state then the machine will halt; other models require all entries to be filled.
4. A **state register** that stores the state of the Turing machine, one of finitely many. There is one special *start state* with which the state register is initialized. These states, writes Turing, replace the "state of mind" a person performing computations would ordinarily be in.



The head is always over a particular square of the tape; only a finite stretch of squares is given. The instruction to be performed (q_4) is shown over the scanned square. (Drawing after Kleene (1952) p.375.)



Here, the internal state (q_1) is shown inside the head, and the illustration describes the tape as being infinite and pre-filled with "0", the symbol serving as blank. The system's full state (its *configuration*) consists of the internal state, the contents of the shaded squares including the blank scanned by the head ("11B"), and the position of the head. (Drawing after Minsky (1967) p. 121).

Note that every part of the machine—its state and symbol-collections—and its actions—printing, erasing and tape motion—is *finite*, *discrete* and *distinguishable*; it is the potentially unlimited amount of tape that gives it an unbounded amount of storage space.

Examples of Turing machines

To see examples of the following models, see Turing machine examples:

1. Turing's very first machine
2. Copy routine
3. 3-state busy beaver

Formal definition

Hopcroft and Ullman (1979, p. 148) formally define a (one-tape) Turing machine as a 7-tuple $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$ where

- Q is a finite, non-empty set of *states*
- Γ is a finite, non-empty set of the *tape alphabet/symbols*
- $b \in \Gamma$ is the *blank symbol* (the only symbol allowed to occur on the tape infinitely often at any step during the computation)
- $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of *input symbols*
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is the set of *final* or *accepting states*.
- $\delta : Q \setminus F \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a partial function called the *transition function*, where L is left shift, R is right shift. (A relatively uncommon variant allows "no shift", say N, as a third element of the latter set.)

Anything that operates according to these specifications is a Turing machine.

The 7-tuple for the 3-state busy beaver looks like this (see more about this busy beaver at Turing machine examples):

Q = { **A, B, C, HALT** }
Γ = { 0, 1 }
b = 0 = "blank"
Σ = { 1 }
δ = see state-table below
q_0 = **A** = initial state
F = the one element set of final states {**HALT**}

Initially all tape cells are marked with 0.

↓ they never introduced states well

State table for 3 state, 2 symbol busy beaver

Tape symbol	Current state A			Current state B			Current state C		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state
0	1	R	**B**	1	L	**A**	1	L	**B**
1	1	L	**C**	1	R	**B**	1	R	**HALT**

Additional details required to visualize or implement Turing machines

In the words of van Emde Boas (1990), p. 6: "The set-theoretical object [his formal seven-tuple description similar to the above] provides only partial information on how the machine will behave and what its computations will look like."

For instance,

- There will need to be some decision on what the symbols actually look like, and a failproof way of reading and writing symbols indefinitely.
- The shift left and shift right operations may shift the tape head across the tape, but when actually building a Turing machine it is more practical to make the tape slide back and forth under the head instead. 6.042
- The tape can be finite, and automatically extended with blanks as needed (which is closest to the mathematical definition), but it is more common to think of it as stretching infinitely at both ends and being pre-filled with blanks except on the explicitly given finite fragment the tape head is on. (This is, of course, not implementable in practice.) The tape *cannot* be fixed in length, since that would not correspond to the given definition and would seriously limit the range of computations the machine can perform to those of a linear bounded automaton.

duh ?

Alternative definitions

Definitions in literature ~~sometimes differ slightly, to~~ make arguments or proofs easier or clearer, but this is always done in such a way that the resulting machine has the same computational power. For example, changing the set $\{L,R\}$ to $\{L,R,N\}$, where N ("None" or "No-operation") would allow the machine to stay on the same tape cell instead of moving left or right, does not increase the machine's computational power.

The most common convention represents each "Turing instruction" in a "Turing table" by one of nine 5-tuples, per the convention of Turing/Davis (Turing (1936) in *Undecidable*, p. 126-127 and Davis (2000) p. 152):

(definition 1): **(q_i, S_j, S_k/E/N, L/R/N, q_m)**
(current state q_i , symbol scanned S_j , print symbol S_k/erase E/none N , move_tape_one_square left L/right R/none N , new state q_m)

Other authors (Minsky (1967) p. 119, Hopcroft and Ullman (1979) p. 158, Stone (1972) p. 9) adopt a different convention, with new state q_m listed immediately after the scanned symbol S_j:

(definition 2): **(q_i, S_j, q_m, S_k/E/N, L/R/N)**
(current state q_i , symbol scanned S_j , new state q_m , print symbol S_k/erase E/none N , move_tape_one_square left L/right R/none N)

For the remainder of this article "definition 1" (the Turing/Davis convention) will be used.

how prove that can compute anything

Example: state table for the 3-state 2-symbol busy beaver reduced to 5-tuples

Current state	Scanned symbol	Print symbol	Move tape	Final (i.e. next) state	5-tuples
A	0	1	R	B	(A, 0, 1, R, B)
A	1	1	L	C	(A, 1, 1, L, C)
B	0	1	L	A	(B, 0, 1, L, A)
B	1	1	R	B	(B, 1, 1, R, B)
C	0	1	L	B	(C, 0, 1, L, B)
C	1	1	N	H	(C, 1, 1, N, H)

In the following table, Turing's original model allowed only the first three lines that he called N1, N2, N3 (cf Turing in *Undecidable*, p. 126). He allowed for erasure of the "scanned square" by naming a 0th symbol S_0 = "erase" or "blank", etc. However, he did not allow for non-printing, so every instruction-line includes "print symbol S_k" or "erase" (cf footnote 12 in Post (1947), *Undecidable* p. 300). The abbreviations are Turing's (*Undecidable* p. 119). Subsequent to Turing's original paper in 1936–1937, machine-models have allowed all nine possible types of five-tuples:

	Current m-configuration (Turing state)	Tape symbol	Print-operation	Tape-motion	Final m-configuration (Turing state)	5-tuple	5-tuple comments	4-tuple
N1	q_i	S_j	Print(S_k)	Left L	q_m	(q_i, S_j, S_k, L, q_m)	"blank" = S_0, 1=S_1, etc.	
N2	q_i	S_j	Print(S_k)	Right R	q_m	(q_i, S_j, S_k, R, q_m)	"blank" = S_0, 1=S_1, etc.	
N3	q_i	S_j	Print(S_k)	None N	q_m	(q_i, S_j, S_k, N, q_m)	"blank" = S_0, 1=S_1, etc.	(q_i, S_j, S_k, q_m)
4	q_i	S_j	None N	Left L	q_m	(q_i, S_j, N, L, q_m)		(q_i, S_j, L, q_m)
5	q_i	S_j	None N	Right R	q_m	(q_i, S_j, N, R, q_m)		(q_i, S_j, R, q_m)
6	q_i	S_j	None N	None N	q_m	(q_i, S_j, N, N, q_m)	Direct "jump"	(q_i, S_j, N, q_m)

7	q_i	S_j	Erase	Left L	q_m	(q_i, S_j, E, L, q_m)		
8	q_i	S_j	Erase	Right R	q_m	(q_i, S_j, E, R, q_m)		
9	q_i	S_j	Erase	None N	q_m	(q_i, S_j, E, N, q_m)		(q_i, S_j, E, q_m)

Any Turing table (list of instructions) can be constructed from the above nine 5-tuples. For technical reasons, the three non-printing or "N" instructions (4, 5, 6) can usually be dispensed with. For examples see Turing machine examples.

Less frequently the use of 4-tuples are encountered: these represent a further atomization of the Turing instructions (cf Post (1947), Boolos & Jeffrey (1974, 1999), Davis-Sigal-Weyuker (1994)); also see more at Post–Turing machine.

The "state"

The word "state" used in context of Turing machines can be a source of confusion, as it can mean two things. Most commentators after Turing have used "state" to mean the name/designator of the current instruction to be performed—i.e. the contents of the state register. But Turing (1936) made a strong distinction between a record of what he called the machine's "m-configuration", (its internal state) and the machine's (or person's) "state of progress" through the computation - the current state of the total system. What Turing called "the state formula" includes both the current instruction and *all* the symbols on the tape:

> Thus the state of progress of the computation at any stage is completely determined by the note of instructions and the symbols on the tape. That is, the **state of the system** may be described by a single expression (sequence of symbols) consisting of the symbols on the tape followed by Δ (which we suppose not to appear elsewhere) and then by the note of instructions. This expression is called the 'state formula'.
> —*Undecidable*, p.139–140, emphasis added

SM not part of formal def in public?

Earlier in his paper Turing carried this even further: he gives an example where he places a symbol of the current "m-configuration"—the instruction's label—beneath the scanned square, together with all the symbols on the tape (*Undecidable*, p. 121); this he calls "the *complete configuration*" (*Undecidable*, p. 118). To print the "complete configuration" on one line he places the state-label/m-configuration to the *left* of the scanned symbol.

A variant of this is seen in Kleene (1952) where Kleene shows how to write the Gödel number of a machine's "situation": he places the "m-configuration" symbol q_4 over the scanned square in roughly the center of the 6 non-blank squares on the tape (see the Turing-tape figure in this article) and puts it to the *right* of the scanned square. But Kleene refers to "q_4" itself as "the machine state" (Kleene, p. 374-375). Hopcroft and Ullman call this composite the "instantaneous description" and follow the Turing convention of putting the "current state" (instruction-label, m-configuration) to the *left* of the scanned symbol (p. 149).

Example: total state of 3-state 2-symbol busy beaver after 3 "moves" (taken from example "run" in the figure below):

1A1

This means: after three moves the tape has ... 000110000 ... on it, the head is scanning the right-most 1, and the state is **A**. Blanks (in this case represented by "0"s) can be part of the total state as shown here: **B**01 ; the tape has a single 1 on it, but the head is scanning the 0 ("blank") to its left and the state is **B**.

"State" in the context of Turing machines should be clarified as to which is being described: (*i*) the current instruction, or (*ii*) the list of symbols on the tape together with the current instruction, or (*iii*) the list of symbols on the tape together with the current instruction placed to the left of the scanned symbol or to the right of the scanned symbol.

Turing's biographer Andrew Hodges (1983: 107) has noted and discussed this confusion.

Turing machine "state" diagrams

The table for the 3-state busy beaver ("P" = print/write a "1")

Tape symbol	Current state A			Current state B			Current state C		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state

0	P	R	**B**	P	L	**A**	P	L	**B**
1	P	L	**C**	P	R	**B**	P	R	**HALT**

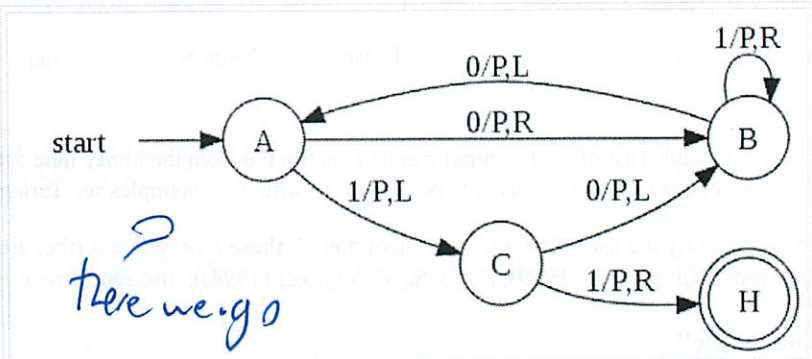
To the right: the above TABLE as expressed as a "state transition" diagram.

Usually large TABLES are better left as tables (Booth, p. 74). They are more readily simulated by computer in tabular form (Booth, p. 74). However, certain concepts—e.g. machines with "reset" states and machines with repeating patterns (cf Hill and Peterson p. 244ff)—can be more readily seen when viewed as a drawing.

Whether a drawing represents an improvement on its TABLE must be decided by the reader for the particular context. See Finite state machine for more. yeah

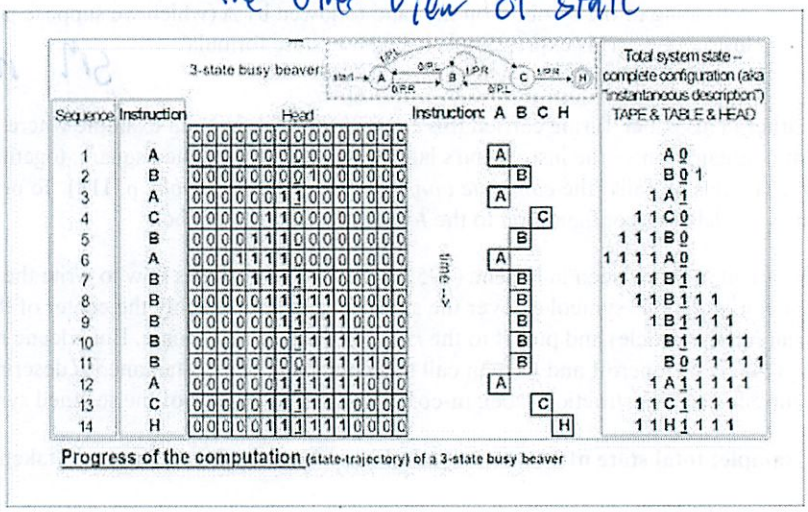
The reader should again be cautioned that such diagrams represent a snapshot of their TABLE frozen in time, *not* the course ("trajectory") of a computation *through* time and/or space. While every time the busy beaver machine "runs" it will always follow the same state-trajectory, this is not true for the "copy" machine that can be provided with variable input "parameters".

The diagram "Progress of the computation" shows the 3-state busy beaver's "state" (instruction) progress through its computation from start to finish. On the far right is the Turing "complete configuration" (Kleene "situation", Hopcroft–Ullman "instantaneous description") at each step. If the machine were to be stopped and cleared to blank both the "state register" and entire tape, these "configurations" could be used to rekindle a computation anywhere in its progress (cf Turing (1936) *Undecidable* pp. 139–140).



The "3-state busy beaver" Turing Machine in a finite state representation. Each circle represents a "state" of the TABLE—an "m-configuration" or "instruction". "Direction" of a state *transition* is shown by an arrow. The label (e.g.. **0/P,R**) near the outgoing state (at the "tail" of the arrow) specifies the scanned symbol that causes a particular transition (e.g. **0**) followed by a slash /, followed by the subsequent "behaviors" of the machine, e.g. "**P** Print" then move tape "**R** Right". No general accepted format exists. The convention shown is after McClusky (1965), Booth (1967), Hill and Peterson (1974).

the other view of state



The evolution of the busy-beaver's computation starts at the top and proceeds to the bottom.

Models equivalent to the Turing machine model

See also: Turing machine equivalents, Register machine, and Post–Turing machine

Many machines that might be thought to have more computational capability than a simple universal Turing machine can be shown to have no more power (Hopcroft and Ullman p. 159, cf Minsky (1967)). They might compute faster, perhaps, or use less memory, or their instruction set might be smaller, but they cannot compute more powerfully (i.e. more mathematical functions). (Recall that the Church–Turing thesis *hypothesizes* this to be true for any kind of machine: that anything that can be "computed" can be computed by some Turing machine.)

A Turing machine is equivalent to a pushdown automaton that has been made more flexible and concise by relaxing the last-in-first-out requirement of its stack. oh cool

At the other extreme, some very simple models turn out to be Turing-equivalent, i.e. to have the same computational power as the Turing machine model. – just like a stack

Common equivalent models are the multi-tape Turing machine, multi-track Turing machine, machines with input and output, and the

probability

non-deterministic Turing machine (NDTM) as opposed to the *deterministic* Turing machine (DTM) for which the action table has at most one entry for each combination of symbol and state.

Read-only, right-moving Turing Machines are equivalent to NDFA's (as well as DFA's by conversion using the NDFA to DFA conversion algorithm).

For practical and didactical intentions the equivalent register machine can be used as a usual assembly programming language.

Choice c-machines, Oracle o-machines

Early in his paper (1936) Turing makes a distinction between an "automatic machine"—its "motion ... completely determined by the configuration" and a "choice machine":

> ...whose motion is only partially determined by the configuration ... When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator. This would be the case if we were using machines to deal with axiomatic systems.
> —*Undecidable*, p. 118

Turing (1936) does not elaborate further except in a footnote in which he describes how to use an a-machine to "find all the provable formulae of the [Hilbert] calculus" rather than use a choice machine. He "suppose[s] that the choices are always between two possibilities 0 and 1. Each proof will then be determined by a sequence of choices $i_1, i_2, ..., i_n$ ($i_1 = 0$ or 1, $i_2 = 0$ or 1, ..., $i_n = 0$ or 1), and hence the number $2^n + i_1 2^{n-1} + i_2 2^{n-2} + ... + i_n$ completely determines the proof. The automatic machine carries out successively proof 1, proof 2, proof 3, ..." (Footnote ‡, *Undecidable*, p. 138)

This is indeed the technique by which a deterministic (i.e. a-) Turing machine can be used to mimic the action of a nondeterministic Turing machine; Turing solved the matter in a footnote and appears to dismiss it from further consideration.

An oracle machine or o-machine is a Turing a-machine that pauses its computation at state "o" while, to complete its calculation, it "awaits the decision" of "the oracle"—an unspecified entity "apart from saying that it cannot be a machine" (Turing (1939), Undecidable p. 166–168). The concept is now actively used by mathematicians.

Universal Turing machines

Main article: Universal Turing machine

As Turing wrote in *Undecidable*, p. 128 (italics added):

programmable

> It is possible to invent a *single machine* which can be used to compute *any* computable sequence. If this machine **U** is supplied with the tape on the beginning of which is written the string of quintuples separated by semicolons of some computing machine **M**, then **U** will compute the same sequence as **M**.

This finding is now taken for granted, but at the time (1936) it was considered astonishing. The model of computation that Turing called his "universal machine"—"U" for short—is considered by some (cf Davis (2000)) to have been the fundamental theoretical breakthrough that led to the notion of the Stored-program computer.

> Turing's paper ... contains, in essence, the invention of the modern computer and some of the programming techniques that accompanied it.
> —Minsky (1967), p. 104

In terms of computational complexity, a multi-tape universal Turing machine need only be slower by logarithmic factor compared to the machines it simulates. This result was obtained in 1966 by F. C. Hennie and R. E. Stearns. (Arora and Barak, 2009, theorem 1.9)

Comparison with real machines

its just like his guy
-except instead of having diff guys for each calc-
tell guy which calc to do

It is often said that Turing machines, unlike simpler automata, are as powerful as real machines, and are able to execute any operation that a real program can. What is missed in this statement is that, because a real machine can only be in finitely many *configurations*, in fact this "real machine" is nothing but a linear bounded automaton. On the other hand, Turing machines are equivalent to machines that have an unlimited amount of storage space for their computations. In fact, Turing machines are not intended to model computers, but rather they are intended to model computation itself; historically, computers, which compute only on their (fixed) internal storage, were developed only later.

There are a number of ways to explain why Turing machines are useful models of real computers:

1. Anything a real computer can compute, a Turing machine can also compute. For example: "A Turing machine can simulate any type of subroutine found in programming languages, including recursive procedures and any of the known parameter-passing mechanisms" (Hopcroft and Ullman p. 157). A large enough FSA can also model any real computer, disregarding IO. Thus, a statement about the limitations of Turing machines will also apply to real computers. ∞ size!
2. The difference lies only with the ability of a Turing machine to manipulate an unbounded amount of data. However, given a finite amount of time, a Turing machine (like a real machine) can only manipulate a finite amount of data. 6.042 idea
3. Like a Turing machine, a real machine can have its storage space enlarged as needed, by acquiring more disks or other storage media. If the supply of these runs short, the Turing machine may become less useful as a model. But the fact is that neither Turing machines nor real machines need astronomical amounts of storage space in order to perform useful computation. The processing time required is usually much more of a problem. style of thinking
4. Descriptions of real machine programs using simpler abstract models are often much more complex than descriptions using Turing machines. For example, a Turing machine describing an algorithm may have a few hundred states, while the equivalent deterministic finite automaton on a given real machine has quadrillions. This makes the DFA representation infeasible to analyze.
5. Turing machines describe algorithms independent of how much memory they use. There is a limit to the memory possessed by any current machine, but this limit can rise arbitrarily in time. Turing machines allow us to make statements about algorithms which will (theoretically) hold forever, regardless of advances in *conventional* computing machine architecture.
6. Turing machines simplify the statement of algorithms. Algorithms running on Turing-equivalent abstract machines are usually more general than their counterparts running on real machines, because they have arbitrary-precision data types available and never have to deal with unexpected conditions (including, but not limited to, running out of memory).

One way in which Turing machines are a poor model for programs is that many real programs, such as operating systems and word processors, are written to receive unbounded input over time, and therefore do not halt. Turing machines do not model such ongoing computation well (but can still model portions of it, such as individual procedures).

Limitations of Turing machines

Computational Complexity Theory

Further information: Computational complexity theory

A limitation of Turing Machines is that they do not model the strengths of a particular arrangement well. For instance, modern stored-program computers are actually instances of a more specific form of abstract machine known as the random access stored program machine or RASP machine model. Like the Universal Turing machine the RASP stores its "program" in "memory" external to its finite-state machine's "instructions". Unlike the Universal Turing Machine, the RASP has an infinite number of distinguishable, numbered but unbounded "registers"—memory "cells" that can contain any integer (cf. Elgot and Robinson (1964), Hartmanis (1971), and in particular Cook-Rechow (1973); references at random access machine). The RASP's finite-state machine is equipped with the capability for indirect addressing (e.g. the contents of one register can be used as an address to specify another register); thus the RASP's "program" can address any register in the register-sequence. The upshot of this distinction is that there are computational optimizations that can be performed based on the memory indices, which are not possible in a general Turing Machine; thus when Turing Machines are used as the basis for bounding running times, a 'false lower bound' can be proven on certain algorithms' running times (due to the false simplifying assumption of a Turing Machine). An example of this is binary search, an algorithm that can be shown to perform more quickly when using the RASP model of computation rather than the Turing machine model.

diff things - otherwise state table is (# registers)^2!

Concurrency

Another limitation of Turing machines is that they do not model concurrency well. For example, there is a bound on the size of integer that can be computed by an always-halting nondeterministic Turing Machine starting on a blank tape. (See article on Unbounded nondeterminism.) By contrast, there are always-halting concurrent systems with no inputs that can compute an integer of unbounded size. (A process can be created with local storage that is initialized with a count of 0 that concurrently sends itself both a stop and a go message. When it receives a go message, it increments its count by 1 and sends itself a go message. When it receives a stop message, it stops with an unbounded number in its local storage.)

History

See also: Algorithm and Church–Turing thesis

They were described in 1936 by Alan Turing.

Historical background: computational machinery

Robin Gandy (1919–1995)—a student of Alan Turing (1912–1954) and his life-long friend—traces the lineage of the notion of "calculating machine" back to Babbage (circa 1834) and actually proposes "Babbage's Thesis":

> *That the whole of development and operations of analysis are now capable of being executed by machinery.*
> —(italics in Babbage as cited by Gandy, p. 54)

physical

Gandy's analysis of Babbage's Analytical Engine describes the following five operations (cf p. 52–53):

1. The arithmetic functions +, −, × where − indicates "proper" subtraction $x - y = 0$ if $y \geq x$
2. Any sequence of operations is an operation
3. Iteration of an operation (repeating n times an operation P)
4. Conditional iteration (repeating n times an operation P conditional on the "success" of test T)
5. Conditional transfer (i.e. conditional "goto")

Did anyone actually build?

Gandy states that "the functions which can be calculated by (1), (2), and (4) are precisely those which are Turing computable." (p. 53). He cites other proposals for "universal calculating machines" included those of Percy Ludgate (1909), Leonardo Torres y Quevedo (1914), Maurice d'Ocagne (1922), Louis Couffignal (1933), Vannevar Bush (1936), Howard Aiken (1937). However:

> ... the emphasis is on programming a fixed iterable sequence of arithmetical operations. The fundamental importance of conditional iteration and conditional transfer for a general theory of calculating machines is not recognized ...
> —Gandy p. 55

The Entscheidungsproblem (the "decision problem"): Hilbert's tenth question of 1900

With regards to Hilbert's problems posed by the famous mathematician David Hilbert in 1900, an aspect of problem #10 had been floating about for almost 30 years before it was framed precisely. Hilbert's original expression for #10 is as follows:

> **10. Determination of the solvability of a Diophantine equation**. Given a Diophantine equation with any number of unknown quantities and with rational integral coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers. The Entscheidungsproblem [decision problem for first-order logic] is solved when we know a procedure that allows for any given logical expression to decide by finitely many operations its validity or satisfiability ... The Entscheidungsproblem must be considered the main problem of mathematical logic.
> —quoted, with this translation and the original German, in Dershowitz and Gurevich, 2008

By 1922, this notion of "Entscheidungsproblem" had developed a bit, and H. Behmann stated that

> ... most general form of the Entscheidungsproblem [is] as follows:
>
> > A quite definite generally applicable prescription is required which will allow one to decide in a finite number of steps the truth or falsity of a given purely logical assertion ...
> >
> > —Gandy p. 57, quoting Behmann
>
> Behmann remarks that ... the general problem is equivalent to the problem of deciding which mathematical propositions are true.
> —*ibid.*
>
> If one were able to solve the Entscheidungsproblem then one would have a "procedure for solving many (or even all) mathematical problems".
> —*ibid.*, p. 92

By the 1928 international congress of mathematicians Hilbert "made his questions quite precise. First, was mathematics *complete* ... Second, was mathematics *consistent* ... And thirdly, was mathematics *decidable*?" (Hodges p. 91, Hawking p. 1121). The first two questions were answered in 1930 by Kurt Gödel at the very same meeting where Hilbert delivered his retirement speech (much to the chagrin of Hilbert); the third—the Entscheidungsproblem—had to wait until the mid-1930s.

The problem was that an answer first required a precise definition of "*definite general applicable prescription*", which Princeton professor Alonzo Church would come to call "effective calculability", and in 1928 no such definition existed. But over the next 6–7 years Emil Post developed his definition of a worker moving from room to room writing and erasing marks per a list of instructions (Post 1936), as did Church and his two students Stephen Kleene and J. B. Rosser by use of Church's lambda-calculus and Gödel's recursion theory (1934). Church's paper (published 15 April 1936) showed that the Entscheidungsproblem was indeed "undecidable"

and beat Turing to the punch by almost a year (Turing's paper submitted 28 May 1936, published January 1937). In the meantime, Emil Post submitted a brief paper in the fall of 1936, so Turing at least had priority over Post. While Church refereed Turing's paper, Turing had time to study Church's paper and add an Appendix where he sketched a proof that Church's lambda-calculus and his machines would compute the same functions.

> But what Church had done was something rather different, and in a certain sense weaker. ... the Turing construction was more direct, and provided an argument from first principles, closing the gap in Church's demonstration.
> —Hodges p. 112

And Post had only proposed a definition of calculability and criticized Church's "definition", but had proved nothing.

Alan Turing's a- (automatic-)machine

In the spring of 1935 Turing as a young Master's student at King's College Cambridge, UK, took on the challenge; he had been stimulated by the lectures of the logician M. H. A. Newman "and learned from them of Gödel's work and the Entscheidungsproblem ... Newman used the word 'mechanical' ... In his obituary of Turing 1955 Newman writes:

> To the question 'what is a "mechanical" process?' Turing returned the characteristic answer 'Something that can be done by a machine' and he embarked on the highly congenial task of analysing the general notion of a computing machine.
> —Gandy, p. 74

Gandy states that:

> I suppose, but do not know, that Turing, right from the start of his work, had as his goal a proof of the undecidability of the Entscheidungsproblem. He told me that the 'main idea' of the paper came to him when he was lying in Grantchester meadows in the summer of 1935. The 'main idea' might have either been his analysis of computation or his realization that there was a universal machine, and so a diagonal argument to prove unsolvability.
> —*ibid.*, p. 76

While Gandy believed that Newman's statement above is "misleading", this opinion is not shared by all. Turing had a life-long interest in machines: "Alan had dreamt of inventing typewriters as a boy; [his mother] Mrs. Turing had a typewriter; and he could well have begun by asking himself what was meant by calling a typewriter 'mechanical'" (Hodges p. 96). While at Princeton pursuing his PhD, Turing built a Boolean-logic multiplier (see below). His PhD thesis, titled "Systems of Logic Based on Ordinals", contains the following definition of "a computable function":

> It was stated above that 'a function is effectively calculable if its values can be found by some purely mechanical process'. We may take this statement literally, understanding by a purely mechanical process one which could be carried out by a machine. It is possible to give a mathematical description, in a certain normal form, of the structures of these machines. The development of these ideas leads to the author's definition of a computable function, and to an identification of computability with effective calculability. It is not difficult, though somewhat laborious, to prove that these three definitions [the 3rd is the λ-calculus] are equivalent.
> —Turing (1939) in *The Undecidable*, p. 160

When Turing returned to the UK he ultimately became jointly responsible for breaking the German secret codes created by encryption machines called "The Enigma"; he also became involved in the design of the ACE (Automatic Computing Engine), "[Turing's] ACE proposal was effectively self-contained, and its roots lay not in the EDVAC [the USA's initiative], but in his own universal machine" (Hodges p. 318). Arguments still continue concerning the origin and nature of what has been named by Kleene (1952) Turing's Thesis. But what Turing *did prove* with his computational-machine model appears in his paper *On Computable Numbers, With an Application to the Entscheidungsproblem* (1937):

> [that] the Hilbert Entscheidungsproblem can have no solution ... I propose, therefore to show that there can be no general process for determining whether a given formula U of the functional calculus K is provable, i.e. that there can be no machine which, supplied with any one U of these formulae, will eventually say whether U is provable.
> —from Turing's paper as reprinted in *The Undecidable*, p. 145

Turing's example (his second proof): If one is to ask for a general procedure to tell us: "Does this machine ever print 0", the question is "undecidable".

1937–1970: The "digital computer", the birth of "computer science"

In 1937, while at Princeton working on his PhD thesis, Turing built a digital (Boolean-logic) multiplier from scratch, making his own electromechanical relays (Hodges p. 138). "Alan's task was to embody the logical design of a Turing machine in a network of relay-operated switches ..." (Hodges p. 138). While Turing might have been just curious and experimenting, quite-earnest work in the same

direction was going in Germany (Konrad Zuse (1938)), and in the United States (Howard Aiken) and George Stibitz (1937); the fruits of their labors were used by the Axis and Allied military in World War II (cf Hodges p. 298–299). In the early to mid-1950s Hao Wang and Marvin Minsky reduced the Turing machine to a simpler form (a precursor to the Post-Turing machine of Martin Davis); simultaneously European researchers were reducing the new-fangled electronic computer to a computer-like theoretical object equivalent to what was now being called a "Turing machine". In the late 1950s and early 1960s, the coincidentally-parallel developments of Melzak and Lambek (1961), Minsky (1961), and Shepherdson and Sturgis (1961) carried the European work further and reduced the Turing machine to a more friendly, computer-like abstract model called the counter machine; Elgot and Robinson (1964), Hartmanis (1971), Cook and Reckhow (1973) carried this work even further with the register machine and random access machine models—but basically all are just multi-tape Turing machines with an arithmetic-like instruction set.

1970–present: the Turing machine as a model of computation

Today the counter, register and random-access machines and their sire the Turing machine continue to be the models of choice for theorists investigating questions in the theory of computation. In particular, computational complexity theory makes use of the Turing machine:

> Depending on the objects one likes to manipulate in the computations (numbers like nonnegative integers or alphanumeric strings), two models have obtained a dominant position in machine-based complexity theory:
>
> > *the off-line multitape Turing machine*..., which represents the standard model for string-oriented computation, and the *random access machine (RAM)* as introduced by Cook and Reckhow ..., which models the idealized Von Neumann style computer.
> >
> > —van Emde Boas 1990:4
>
> Only in the related area of analysis of algorithms this role is taken over by the RAM model.
> —van Emde Boas 1990:16

Kantorovitz (2005),[4] was the first to show the most simple obvious representation of Turing Machines published academically which unifies Turing Machines with mathematical analysis and analog computers.

See also

- Algorithm, for a brief history of some of the inventions and the mathematics leading to Turing's definition of what he called his "a-machine"
- Arithmetical hierarchy
- Bekenstein bound; Because they have an infinite tape, Turing machines are physically impossible if they are to have a finite size and bounded energy.
- BlooP and FlooP
- Busy beaver
- Chaitin constant or Omega (computer science) for information relating to the halting problem
- Church-Turing thesis, which says Turing machines can perform any computation that can be performed
- Conway's Game of Life, a Turing-complete cellular automaton
- Genetix a virtual machine created by Bernard Hodson containing only 34 executable instructions.
- *Gödel, Escher, Bach: An Eternal Golden Braid*, an influential book largely about the Church-Turing Thesis.
- Halting problem, for more references
- Harvard architecture
- Hyperbrain a theoretical model of the brain
- Langton's ant and Turmites, simple two-dimensional analogues of the Turing machine.
- Modified Harvard architecture
- Probabilistic Turing machine
- Quantum Turing machine
- Turing completeness, an attribute used in computability theory to describe computing systems with power equivalent to a universal Turing machine.
- Turing switch
- Turing tarpit, any computing system or language which, despite being Turing complete, is generally considered useless for practical computing.
- Von Neumann architecture

Notes

1. ^ The idea came to him in mid-1935 (perhaps, see more in the History section) after a question posed by M. H. A. Newman in his lectures -- "Was there a definite method, or as Newman put it, a *mechanical process* which could be applied to a mathematical statement, and which would come up with the answer as to whether it was provable" (Hodges 1983:93). Turing submitted his paper on 31 May 1936 to the London Mathematical Society for its *Proceedings* (cf Hodges 1983:112), but it was *published* in early 1937 -- offprints available February 1937 (cf Hodges 1983:129).

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Lab #4

Preparation: the description of Turing Machines in Lecure #10 "Models of Computation" will be useful when working on this lab.

Turing Machine Simulation: TMSim

Need to read about Turing machines

The goal of this lab is write the FSM controller for a Turing Machine (TM) which checks to see if the string of left and right parentheses it finds on its input tape "balance".

The TM has a doubly-infinite tape with discrete symbol positions (cells) each of which contains one of a finite set of symbols. The control FSM has one input: the symbol found in the current cell . The FSM produces several outputs: the symbol to be written into the current cell and a motion control that determines how the head should move. In our simulation, the tape is displayed horizontally, so the tape can move left, right, or stay where it is.

The operation of the TM is specified by a file containing one or more of the following statements:

`// comment`

C++-style comment: ignore characters starting with the '//' and continuing to the end of the current line.

`/* .... */`

C-style comment: ignore characters between "/*" and "*/". Note that the ignored characters may include newlines; this type of comment can be used to comment-out multiple lines of your file.

`symbols` *`symbol`*...

Declare one or more tape symbols. The symbol "–" (dash) is predefined and is used to indicate that a tape cell is blank. You have to declare symbols you use in an action statement (see below). A symbol can be any sequence of non-whitespace characters not including "/" or the quote character. If you want to declare a symbol containing whitespace, "/" or quote, you must enclose the symbol in quotes. You can have more than one `symbols` statement in your file.

`states` *`state`*...

Declare one or more states. There are two predefined states: "*halt*" and "*error*". The TM simulation will stop if either of these states is reached. The "*error*" state is useful for indicating that the TM has halted due to an unexpected condition. You can have more than one `states` statement in your file. The first state specified by the first `states` statement is the starting state for the TM.

`action` *`state symbol newstate writesymbol motion`*

Specify the action performed by the TM when the current state is *state* and the current symbol is *symbol*. First the TM will write *writesymbol* into the current cell of the tape. Then the tape is moved left if "`l`" is specified for the motion, right if "`r`" is specified and

remain where it is if "–" is specified. Finally the current state of the control FSM is changed to *newstate* and the TM searches for the next applicable action. If *newstate* is "*halt*" or "*error*", the TM simulation stops. If there is no action specified for the current state and current symbol, the TM enters the "*error*" state. Note that you have to declare any symbols or states you use in an action statement – this requirement is helpful in catching typos.

`tape` *`name symbol…`*

Specifies the initial configuration of a TM tape, each tape has a name. The various names are displayed as a set of radio buttons at the top of the TM animation – you can select which tape is loaded at reset by clicking on one of the buttons. You can specify which cell of the tape is to be current cell after reset by enclosing the appropriate symbol in square brackets. For example, an initial tape configuration called "test" consisting of three non-blank cells with the head positioned over the middle cell is specified by

```
tape test 1 [2] 3
```

If no initial head position is specified, the head is positioned over the leftmost symbol on the tape.

`result` *`name symbol…`*

Specifies the expected head position and contents of the tape after the TM has finished processing the initial tape configuration called *name*. This statement is used by the checkoff system to verify that your TM has correctly processed each of the test tapes. Whenever the TM enters the "*halt*" state, the final tape configuration is checked against the appropriate result statement if one has been specified and any discrepancies will be reported in the status display at the bottom of the TMSim window.

`result1` *`name symbol`*

Like `result` except that only the current symbol is checked against the specified value.

`checkoff` *`server assignment checksum`*

This information is used by TMSim to contact the on-line checkoff server when you invoke the checkoff tool (click the green checkmark in the toolbar). In order to complete the checkoff, you need to have run your TM on all the supplied test tapes and have each of the final configurations match the specified results.

Here's an example file that defines a control FSM with three states (s1, s2 and s3) and two possible tape symbols: "1" and "–" (recall that the "–" symbol is predefined). There is a single tape configuration defined called "test" which consists of a blank tape. The final tape configuration is expected to be a tape containing six consecutive "1" symbols with the head positioned over the second "1". Note that there is an action declared for each possible combination of the three states and two tape symbols.

```
// 3-state busy beaver Turing Machine example

// See how many 1's we can write on a blank tape using
// only a three-state Turing Machine

states s1 s2 s3  // list of state names, first is starting state
```

```
symbols 1          // list of symbols (- is blank cell)

tape test -        // initial tape contents, blank in this case
result test 1 [1] 1 1 1 1 // expected result

// specify transistions: action state symbol state' write move
//     state = the current state of the FSM
//     symbol = the symbol read from the current cell
//     state' = state on the next cycle
//     write = symbol to be written into the current cell
//     move = tape movement ("l"=left, "r"=right, "-"=stay put)
action s1 - s2      1 r
action s1 1 s3      1 l
action s2 - s1      1 l
action s2 1 s2      1 r
action s3 - s2      1 l
action s3 1 *halt* 1 r
```

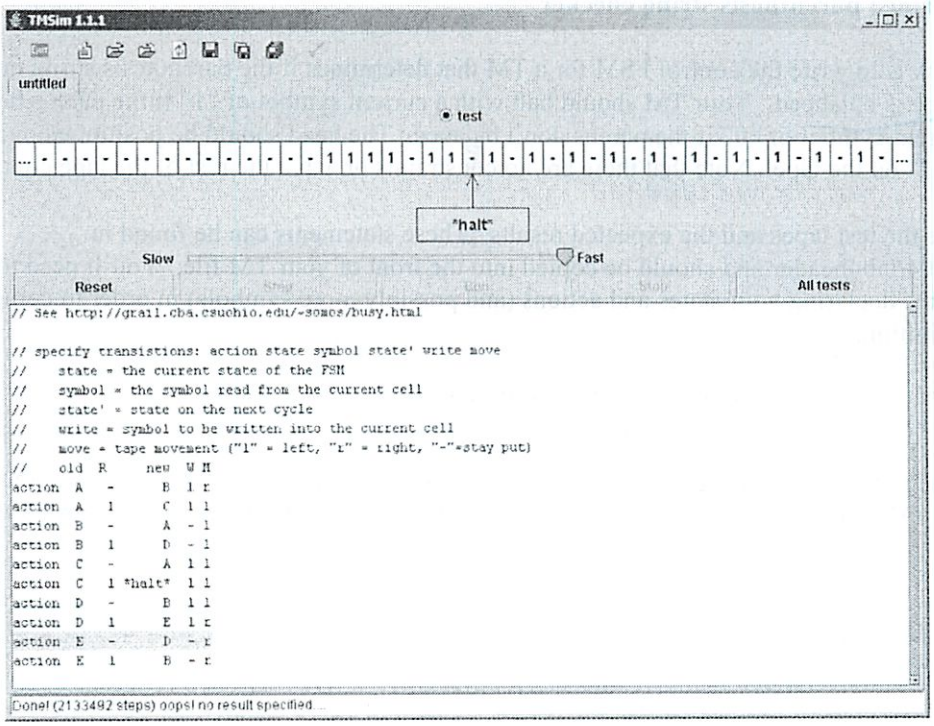
← so if S1 = blank
write 1
go right
this is s2?

* remember state and tape position different!

To run TMSim, login to your Athena account, add the 6.004 locker and type

```
athena% tmsim [filename]
```

You can supply an optional filename argument which will be loaded into the FSM editing buffer (you can load and save FSM files from within TMSim too). If no argument is supplied, the buffer is initialized with a sample FSM. After TMSim starts, you'll see a window with the FSM displayed at the bottom and a state/tape animation at the top.



The TM display consists of the following parts:

Tape select radio buttons. Select which of the named test tapes to use when initializing the TM after reset.

Tape display: Shows the current state and symbol.

Speed control. This slider controls the speed of the animation when you press the "Run" button. At the fastest speed, no status updates are performed which leads to a much faster simulation.

"Reset" button. Reset the TM to its initial state and selected tape configuration.

"Step" button. Let the TM progress one state of the FSM.

"Run" button. Like "Step" except the TM will continue running the FSM until it reaches the "*halt*" or "*error*" state, the "Stop" button is pressed, or an error is detected.

"Stop" button. Stop the TM. You can proceed by pressing the "Step" or "Run" button.

"All tests" button. Automates the task of selecting each test tape in turn and clicking the "Run" button. The automated process will stop if an error is detected.

At the bottom of the screen is a state display showing the current cycle count and any discrepancies detected in the final tape configuration when the TM enters the "*halt*" state.

Tasks

Well-formed parenthesis string checker

Your task is to write the control FSM for a TM that determines if the parenthesis string on its input tape is balanced. Your TM should halt with a current symbol of "1" if the parens balance, or a current symbol of "0" if the parens don't balance. The head should be positioned over the "0" or "1" on the tape. Note that there are no constraints on what the rest of the tape contains.

Here are the test tapes and the expected results. These statements can be found in /mit/6.004/lab4header and should be copied into the front of your TM file. You'll need to add statements declaring your states and actions (and possibly more symbols) in order to complete the TM definition.

```
// Parenthesis matcher Turing Machine
// test tapes and checkoff information

checkoff "6004.csail.mit.edu/currentsemester/6004assignment.doit"
 "Lab #4" 1103641864  // this should be at the end of the line above

symbols ( ) 0 1

tape   test1 (
result1 test1 0

tape   test2 )
result1 test2 0

tape   test3 ( )
```

```
result1 test3 1

tape   test4 ) (
result1 test4 0

tape   test5 ( ) ( ) ( ( ( ) ) ( ) ) ) ( )
result1 test5 0

tape   test6 ( ) ( ( ( ) ( ( ( ) ) ( ) ) )
result1 test6 0

tape   test7 ( ) ( ( ) ( ( ( ) ) ( ) ) )
result1 test7 1

// define additional symbols, your states and actions here…
```

Note that you're welcome to add your own test tapes while debugging your implementation, but you'll need to comment them out before running the checkoff tests (otherwise the checksum mechanism will get confused).

Scoring: The number of points you'll receive is determined by the number of states in your TM definition:

4 points: 2 states
3 points: 3 states
2 points: 4 states
1 point: 5 or more states

try and do it in 2 states

seems easy looks

need to read Turing machine

Lab 4

10/20

Turing machine

Declare states
symbols
tape
actions

Write if parentese are balanced

tests are provided

So do in 2 states

Stuff is on tape already

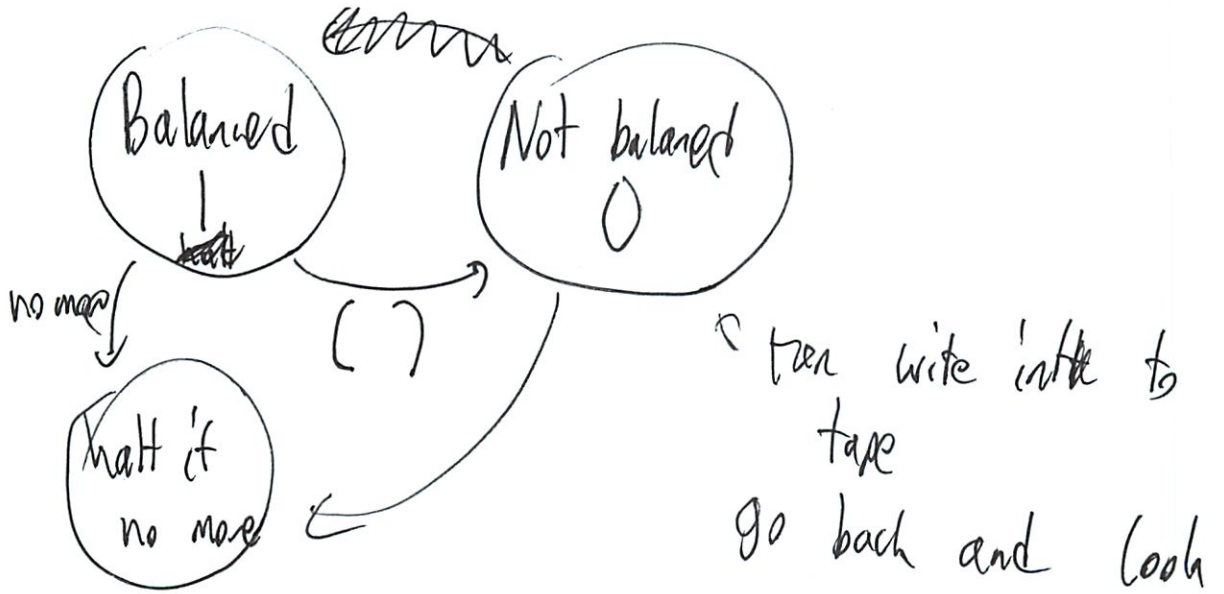
1 if balanced

0 if not

State	Symbol	newstate	writesymbol	motion
↑ current	↑ current			

(2)

So how to do in 2 states?
I would do # off set on each
? Can save to tape



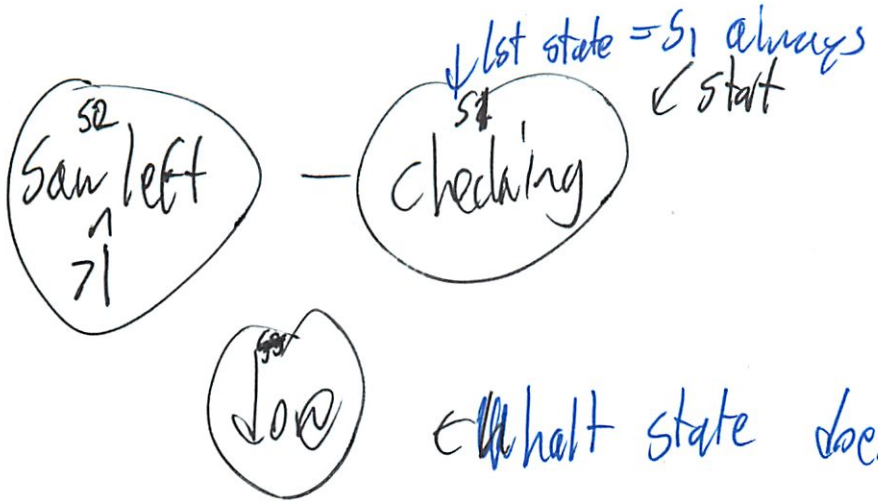
Can't do #s on tape
- must have case for each #

Do 3-state
- one is a checker
  - go back on tape

- Can mark as good - right parent
- Then back to left paren

2 states diff model

(3)

So 

action	S1	(	S2	(	l

← moves tape

action	S1	)	S1	)	l
action	S2	)	S3	–	^

↑ S3 remove

action	S3	)	S3	)	^
action	S3	(	S1	–	l
action	S2	(	S2	(	l

need a halt – when balanced
try this first
Need actions for blank

By – I mean 0

(4) Run it

But we don't halt

and need S2 —

So when we first see —, we should halt
in any state

So make fixed be 0

and blank be —

So first time ~~we~~ S2 or S1 see —, halt
for S3 overshot — so do what? fail (halt)

~~It~~ It should never see → error

So what dir should S2 move in

When halts should see answer in curret cell

So 1 s1 ?
0 S2

Test 1 ✓

~~Or when~~ Test 2 ✗

Or when see — go left till get to end w/o ()

(5) Then done if reach end

(I should think more strategically)
not test by test

Yeah I am off track. Do more states if possible

Move left

Find ≥1 (

Move right

merge

replace symbol
go back right

Move left (S1)

Find ≥1 ((S2)
Move right

S1 (S2 (ℓ
S1) S1) ℓ

S2 (S2 (r
S2) S1 0 ℓ

S1 0 — S2 — r
S2 — "done" 1

but where is fail?

(6)

Try it

(✓) Test 1

action 52 0 "halt" 0

(✓) Test 2

Be careful about what is 1 and 0

~~Record~~ play w/ things on pc

(✓) Test 3

(✓) Test 4 auto works

Test 5

Unexpected step

this is first long op

When we see the 1 again its like we are in special "good" state

but don't want new state

Need other way to see succeeds

Do 53 for now

(7)

I think my ideas of states and directions wrong

So I am failing on ((

When see (, I should go right first and look for)

But that totally messes up my halting system

So (go left mark at least ((s2)

Then in S2 see (keep going left (S2)

in S2 see) then mark go right S3

So rewrite:

Have 4 states now

So gets tests 1-6 ✓

After last) go back and check rest before ending

I should think more about halting

So when

|| (|| ((||) ||))

what should it do?

(8)

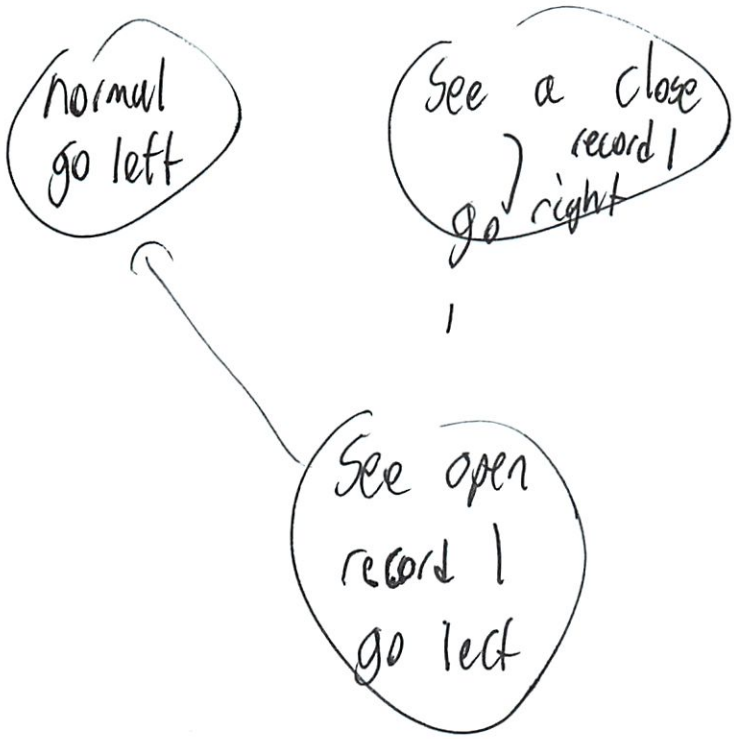
I think this is a big problem

Could do new state → rewind ....

Then if reach end again fail?

No I have way too many states

Use extra symbols



Stash which (working on

Must recheck (

Mark first (- ignore rest

q

Try to find right

(7)

Now go back to q
Mark it 1
keep going left

3 states going left normal
" " q
q1 right

Do halting

I like this q much better!
Got to 5 - still slight error
(✓) No error passed P-set w/ 3 states 3/4 pts
- worked very well

How can I reduce to 2 steps
Challenge problem
From before was told need all new approach

TA: No hints - take away a state
TA: Go do your other hw

(10)

But first check off

(V) Done

A few thoughts on state 2

all are pretty well used

Can we write more to tape?

Don't know moved right or left

; Combine saw q and not

└ what he hinted at

Or don't need a go write state

Write more symbols on tape

- Like wrote C with ~~more symbols~~ q as diff symbols

Can do 2 state w/ just 0 and 1

- Very strange

3 state is most elegant

(11)

So try w/ wrote q as %
That makes no sense
Ah no - give up too tired

Summary of β Instruction Formats

Registers ~~that~~ reserved for Special cases

Register	Symbol	Usage
R31	R31	Always zero
R30	XP	Exception pointer
R29	SP	Stack pointer
R28	LP	Linkage pointer
R27	BP	Base of frame pointer

Operate Class:

31 – 26	25 – 21	20 – 16	15 – 11	10 – 0
10xxxx	Rc	Ra	Rb	*unused*

OP(Ra,Rb,Rc): Reg[Rc] ← Reg[Ra] op Reg[Rb]

Opcodes: **ADD** (plus), **SUB** (minus), **MUL** (multiply), **DIV** (divided by)
AND (bitwise and), **OR** (bitwise or), **XOR** (bitwise exclusive or), **XNOR** (bitwise exclusive nor),
CMPEQ (equal), **CMPLT** (less than), **CMPLE** (less than or equal) [result = 1 if true, 0 if false]
SHL (left shift), **SHR** (right shift w/o sign extension), **SRA** (right shift w/ sign extension)

31 – 26	25 – 21	20 – 16	15 – 0
11xxxx	Rc	Ra	literal (two's complement)

OPC(Ra,literal,Rc): Reg[Rc] ← Reg[Ra] op SEXT(literal)

Opcodes: **ADDC** (plus), **SUBC** (minus), **MULC** (multiply), **DIVC** (divided by)
ANDC (bitwise and), **ORC** (bitwise or), **XORC** (bitwise exclusive or), **XNORC** (bitwise exclusive nor)
CMPEQC (equal), **CMPLTC** (less than), **CMPLEC** (less than or equal) [result = 1 if true, 0 if false]
SHLC (left shift), **SHRC** (right shift w/o sign extension), **SRAC** (right shift w/ sign extension)

Other:

31 – 26	25 – 21	20 – 16	15 – 0
01xxxx	Rc	Ra	literal (two's complement)

LD(Ra,literal,Rc): Reg[Rc] ← Mem[Reg[Ra] + SEXT(literal)]
ST(Rc,literal,Ra): Mem[Reg[Ra] + SEXT(literal)] ← Reg[Rc]
JMP(Ra,Rc): Reg[Rc] ← PC + 4; PC ← Reg[Ra]
BEQ/BF(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] = 0 then PC ← PC + 4 + 4*SEXT(literal)
BNE/BT(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] ≠ 0 then PC ← PC + 4 + 4*SEXT(literal)
LDR(label,Rc): Reg[Rc] ← Mem[PC + 4 + 4*SEXT(literal)]

Opcode Table: (*optional opcodes)

2:0 / *5:3*	*000*	*001*	*010*	*011*	*100*	*101*	*110*	*111*
000								
001								
010								
011	**LD**	**ST**		**JMP**	**BEQ**	**BNE**		**LDR**
100	**ADD**	**SUB**	**MUL***	**DIV***	**CMPEQ**	**CMPLT**	**CMPLE**	
101	**AND**	**OR**	**XOR**	**XNOR**	**SHL**	**SHR**	**SRA**	
110	**ADDC**	**SUBC**	**MULC***	**DIVC***	**CMPEQC**	**CMPLTC**	**CMPLEC**	
111	**ANDC**	**ORC**	**XORC**	**XNORC**	**SHLC**	**SHRC**	**SRAC**	

6.004 Recitation

10/21

β named after DEC's α

Memory LD, ST, LDR

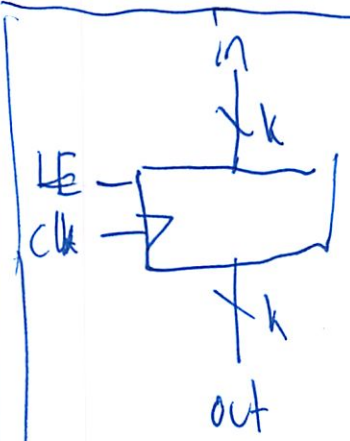
- load, store architecture
  Only instructions that touch main meory

Change PC BNE, BNQ, JMP

Operate $R_c \leftarrow R_b$ op R_a (or) $R_c \leftarrow R_a$ op stx(cnt)

- Do the work
  - ALU does most of this

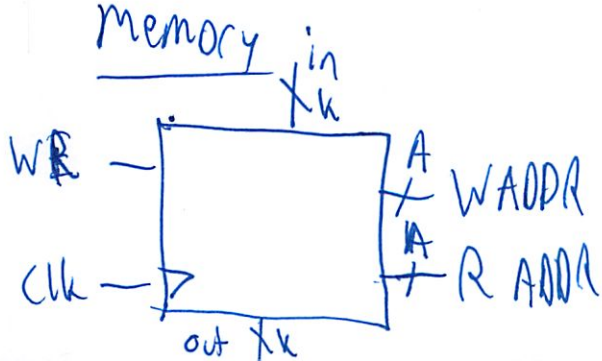
↑ 16 bit constant



k-bit register

- k seperate registers next to each other

memory

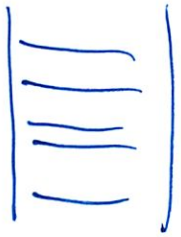


also take addresses

size $2^A \cdot k$ memory

②

Think of this as k-bit registers stacked together



can select <u>which</u> of the registers to read from

└ multiplexer could be used

β actually has <u>2</u> inputs to read 2 at once

Often it uses less ~~registers~~ gates than just multiplexer

└ optimizations reduce # gates needed

So in Beta small amt of memory in registers

└ uses a little memory chip (described above)

have 32 of them

R0, R1, ..., R31

R31 is special - always reads 0

So pass around 5-bit instruction/pointer to the memory

└ call Ra, Rb, Rc

variable named placeholders

③

Q1. Write assembly lang program
Load value from memory whose pointer is called x
32 bit

Do 2 diff things based on side
if $x < 0x8000$

LD (R31, x, R0)

So we load

EA = Reg[R31] + X

EA: effective address
Reg[R31]: look up R31 from Reg (always 0)
X: 16 bit constant

6	5	5	16
Opp	R31	R0	Const

must be $< 0x8000$

0000000	16

pad 0s in front to make 32-bit long

(4)

If x = 0x8000
do something convoluted
make a address w/ memory location

Vaddr : Long (x)
⋮
LDA(Vaddr, R0)
LD(R0, 0, R0)

how has contents of memory

has memory location of x

opps flipped spec

Q2 Write assemply lang for
a = b + 3 x c

Steps
1. Load
2. Operate
3. Store result

a, b, c are stored in memory
know the locations of their memory
which is < 0x8000

before it runs

```
a: Long (0)
b: Long (0)
c: Long (0)
```

← need to define a, b, c

this is defined in the <u>assembler</u> ne

nothing happening on Beta w/ registers or memory

(a, b, c) are locations in memory

load from and store

```
LD(R31, b, R0)
LD(R31, c, R1)
```

→ Can also say LD(c, R1)

We've assumed these locations in memory have been pre filled

```
MVLC(R1, 3, R1)
```

must put c here to say 3 is a constant

← the Rs we put in front of #s are only nice for us. Assembler ignores

```
ADD(R1, R0, R1)
ST(R1, a, R31)
```

← normally compilers manage which registers to use

→ Can also say ST(R1, a)

↑ Assembler converts these instructions to our 32 bit words

6

if (a > b)
  c = 17;

LD(a, R0)
LD(b, R1)
CMPLT (R1, R0, R2) ← b < a

↓ same as above

↳ 0 if false
  1 if true

BF (R2, X123) ← branch if false around instruction

[ADD C (R31, 17, R2)
 ST (R2, C)

could also do
BEQ (R3, X123, R31)
? Branch if R3=0
? where came from

X123: ← leave blank

? the assembler actually converts this to a magic # — that is the address in memory that holds this label

B. uses byte addressing

(7)

Converting assembly lang to actual bits

I = 0x 5768

B = 0x 1234

LD(I, R0) = LD(R31, I, R0)

R_a Const R_c

6 OP	5 RC	5 RA	16 Const
011 00	00000	11111	0101 0111 0110 1000
			5 7 6 8

↑ typically other architectures make R0 to always be 0

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
BSim

Introduction to BSim

BSim is a simulator for the 6.004 Beta architecture. The BSim user interface is very similar to JSim's: there's a simple editor for typing in your program and some tools for assembling the program into binary, loading the program into the simulated Beta's memory, executing the program and examining the results.

To run BSim, login to an Athena console. After signing onto the Athena station, add the 6.004 locker to gain access to the design tools and model files (you only have to do this once each session):

```
athena% add 6.004
```

Start BSim running in a separate window by typing

```
athena% bsim &
```

It can take a few moments for the Java runtime system to start up, please be patient! BSim takes as input a *assembly language program* to be executed. The initial BSim window is a very simple editor that lets you enter and modify your netlist. If you use a separate editor to create your netlists, you can have BSim load your netlist files when it starts:

```
athena% bsim filename … filename &
```

There are various handy buttons on the BSim toolbar:

Exit. Asks if you want to save any modified file buffers and then exits BSim.

New file. Create a new edit buffer called "untitled". Any attempts to save this buffer will prompt the user for a filename.

Open file. Prompts the user for a filename and then opens that file in its own edit buffer. If the file has already been read into a buffer, the buffer will be reloaded from the file (after asking permission if the buffer has been modified).

Close file. Closes the current edit buffer after asking permission if the buffer has been modified.

Reload file. Reload the current buffer from its source file after asking permission if the buffer has been modified. This button is useful if you are using an external

editor to modify the netlist and simply want to reload a new version for simulation.

Save file. If any changes have been made, write the current buffer back to its source file (prompting for a file name if this is an untitled buffer created with the "new file" command). If the save was successful, the old version of the file is saved with a ".bak" extension.

Save file, specifying new file name. Like "Save file" but prompts for a new file name to use.

Save all files. Like "save file" but applied to all edit buffers.

Assemble the current buffer, i.e., convert it into binary and load it into the simulated Beta's memory. Any errors detected will be flagged in the editor window and described in the message area at the bottom of the window. If the assembly completes successfully, a window showing the Beta datapath is created from which you can start execution of the program.

Assemble the current buffer and output the resulting binary to a file whose name is the same as source file for the current buffer with ".bin" appended.

Using information supplied in the checkoff file, check for specified memory values. If all the checks are successful, submit the program to the on-line assignment system.

The Display window has some additional toolbar buttons that are used to control the simulation. The values shown in the window reflect the values on Beta signals after the current instruction has been fetched and executed but just before the register file is updated at the end of the cycle.

Stop execution and update the datapath display.

Reset the contents of the PC and registers to 0, and memory locations to the values they had just after assembly was complete. You have to stop a running simulation before a reset.

Start simulation and run until a HALT() instruction is executed or a breakpoint is reached. You can stop a running simulation using the stop control described above. For maximum simulation speed, the datapath display is not updated until the simulation is stopped.

step by step

Execute the program for a single cycle and then update the display. Very useful for following your program's operation instruction-by-instruction.

Toggle visualization between the programmer's panel (the default) and the animated datapath.

Bring up a window that let's you configure the cache parameters for main memory.

If ".options tty" is specified by the program, a small 5-line typeout window appears at the bottom of the datapath window. You can output characters to this window by executing a WRCHAR() instruction after placing the character value in R0. The tty option also allows for type-in: any character typed by the user causes an interrupt to location 12; RDCHAR() can be used to fetch the character value into R0. Clicking the mouse will cause an interrupts to location 16; CLICK() can be used to fetch the coordinates of the last click into R0. The coordinates are encoded as (x<<16)+y, or -1 if there has been no mouse click since the last call to CLICK(). the consol

If ".options clock" is specified by the program, an interrupt to location 8 is generated every 10,000 cycles. (Remember though that interrupts are disabled until the program enters user mode – see section 6.3 of the Beta documentation.)

Introduction to assembly language

BSim incorporates an *assembler*: a program that converts text files into binary memory data. The simplest assembly language program is a sequence of numerical values which are converted to binary and placed in successive *byte* locations in memory:

```
| Comments begin with vertical bar and end at a newline

37  3   255       | decimal (the default radix)
0b100101          | binary (note the 0b prefix)
0x25              | hexadecimal (note the 0x prefix)
'a'               | character constants
```

Values can also be expressions; e.g., the source file

```
37+0b10-0x10     24 - 0x1  4*0b110-1   0xF7 % 0x20
```

generates 4 bytes of binary output, each with the value 23. Note the operators have no precedence – you have to use parentheses to avoid simple left-to-right evaluation. The available operators are

- unary minus
~ bit-wise complement
+ addition
- subtraction
* multiplication

So assembler does this – not the beta?

does the assembler run on beta?
I don't think so

/	division
%	modulo (result is always positive!)
>>	right shift
<<	left shift

We can also define *symbols* for use in expressions:

```
x = 0x1000          | address in memory of variable X
y = 0x10004         | another address

| Symbolic names for registers
R0 = 0
R1 = 1
...
R31 = 31
```

hard coded in

Note that symbols are case-sensitive: "Foo" and "foo" are different symbols. A special symbol named "." (period) means the address of the next byte to be filled by the assembler:

```
. = 0x100           | assemble into location 0x100
  1  2  3  4
five = .            | symbol five has the value 0x104
  5  6  7  8
. = . + 16          | skip 16 bytes
  9 10 11 12
```

Labels are symbols that represent memory address. They can be set with the following special syntax:

```
X:                  | this is an abbreviation for X = .
```

ok saw in recitation

For example the table on the left shows what main memory will contain after assembling the program on the right.

```
---- MAIN MEMORY ----
  byte:  3  2  1  0

  1000: 09 04 01 00
  1004: 31 24 19 10
  1008: 79 64 51 40
  100C: E1 C4 A9 90
  1010: 00 00 00 10
```

```
. = 0x1000
sqrs:  0 1 4 9
       16 25 36 49
       64 81 100 121
       144 169 196 225
slen:  LONG(. - sqrs)
```

Macros are parameterized abbreviations:

```
| macro to generate 4 consecutive bytes
.macro consec(n) n n+1 n+2 n+3

| invocation of above macro
consec(37)
```

functions?

The macro invocation above has the same effect as

```
37 38 39 40
```

Note that macros evaluate their arguments and substitute the resulting value for occurrences of the corresponding formal parameter in the body of the macro. Here are some macros for breaking multi-byte data into byte-size chunks

```
| assemble into bytes, little-endian format
.macro WORD(x) x%256 (x/256)%256
.macro LONG(x) WORD(x) WORD(x>>16)
LONG(0xdeadbeef)
```

what is this doing?

Has the same effect as

```
0xef 0xbe 0xad 0xde
```

The body of the macro includes the remainder of the line on which the .macro directive appears. Multi-line macros can be defined by enclosing the body in "{" and "}".

beta.uasm contains symbol definitions for all the registers (R0, …, R31, BP, LP, SP, XP, r0, …, r31, bp, lp, sp, xp) and macro definitions for all the Beta instructions:

`OP(Ra,Rb,Rc)`	Reg[Rc] ← Reg[Ra] op Reg[Rb]
Opcodes:	**ADD, SUB, MUL, DIV, AND, OR, XOR**
	CMPEQ, CMPLT, CMPLE, SHL, SHR, SRA
`OPC(Ra,literal,Rc)`	Reg[Rc] ← Reg[Ra] op SEXT($literal_{15:0}$)
Opcodes:	**ADDC, SUBC, MULC, DIVC, ANDC, ORC, XORC**
	CMPEQC, CMPLTC, CMPLEC, SHLC, SHRC, SRAC
`LD(Ra,literal,Rc)`	Reg[Rc] ← Mem[Reg[Ra] + SEXT(literal)]
`ST(Rc,literal,Ra)`	Mem[Reg[Ra] + SEXT(literal)] ← Reg[Rc]
`JMP(Ra,Rc)`	Reg[Rc] ← PC + 4; PC ← Reg[Ra]
`BEQ/BF(Ra,label,Rc)`	Reg[Rc] ← PC + 4; if Reg[Ra] = 0 then PC ← PC + 4 + 4*SEXT(literal)
`BNE/BT(Ra,label,Rc)`	Reg[Rc] ← PC + 4; if Reg[Ra] ≠ 0 then PC ← PC + 4 + 4*SEXT(literal)
`LDR(Ra,label,Rc)`	Reg[Rc] ← Mem[PC + 4 + 4*SEXT(literal)]

Also included are some convenience macros: Shortcuts

```
LD(label,Rc)          expands to LD(R31,label,Rc)
ST(Ra,label)          expands to ST(Ra,label,R31)
BR(label)             expands to BEQ(R31,label,R31)
CALL(label)           expands to BEQ(R31,label,LP)
RTN()                 expands to JMP(LP)
DEALLOCATE(n)         expands to SUBC(SP,n*4,SP)
MOVE(Ra,Rc)           expands to ADD(Ra,R31,Rc)
CMOVE(literal,Rc)     expands to ADDC(R31,literal,Rc)
PUSH(Ra)              expands to ADDC(SP,4,SP) ST(Ra,-4,SP)
POP(Rc)               expands to LD(SP,-4,Rc)  ADDC(SP,-4,SP)
```

2 length

```
HALT()                cause the simulator to stop execution
```

The following is a complete example assembly language program:

```
.include /mit/6.004/bsim/beta.uasm

. = 0                          | start assembling at location 0
    LD(input,r0)               | put argument in r0
    CALL(bitrev)               | call the procedure (= BR(bitrev,r28))
    HALT()

| reverse the bits in r0, leave result in r1
bitrev:
    CMOVE(32,r2)               | loop counter
    CMOVE(0,r1)                | clear output register
loop:
    ANDC(r0,1,r3)              | get low-order bit
    SHLC(r1,1,r1)              | shift output word by 1
    OR(r3,r1,r1)               | OR in new low-order bit
    SHRC(r0,1,r0)              | done with this input bit
    SUBC(r2,1,r2)              | decrement loop counter
    BNE(r2,loop)               | repeat until done
    RTN()                      | return to caller (= JMP(r28))
```

'assembler tracks where to return'

```
input:
    LONG(0x12345)              | 32-bit input (in HEX)
```

The BSim assembly language processor includes a few helpful directives:

`.include filename`
: Process the text found in the specified file at this point in the assembly.

`.align`
`.align expression`
: Increment the value of "." until it is 0 modulo the specified value, e.g., ".align 4" moves to the next word boundary in memory. A value of 4 is used if no expression is given.

`.ascii "chars..."`
: Assemble the characters enclosed in quotes into successive bytes of memory. C-like escapes can be used for non-printing characters.

`.text "chars..."`
: Like .ascii except an additional 0 byte is added to the end of the string in memory.

`.breakpoint`
: Stop the Beta simulator if it fetches an instruction from the current location (i.e., the value of "." at the point the .breakpoint directive occurred). You can define as many breakpoints as you want.

`.protect`
: This directive indicates that subsequent bytes output by the assembler are "protected," causing the simulator to halt if a ST instruction tries to overwrite their value. This directive is useful for protecting code (e.g., the checkoff program) from being overwritten by errant programs.

'how is this implemented?'

`.unprotect`

The opposite of .protect – subsequent bytes output by the assembler are not protected and can be overwritten by the program.

`.options ...`

Used to configure the simulator. Available options:

clk	enable periodic clock interrupts to location 8
noclk	disable clock interrupts (default)
div	simulate the DIV instruction (default)
nodiv	make the DIV opcode an illegal instruction
mul	simulate the MUL instruction (default)
nomul	make the MUL opcode an illegal instruction
kalways	don't let program enter user mode (ie, supervisor bit is always 1)
nokalways	allow program to enter user mode (default)
tty	enable RDCHAR(), WRCHAR(), CLICK() (see end of first section)
notty	RDCHAR(), WRCHAR(), CLICK() are disabled (default)
annotate	if BP is non-zero, label stack frames in the programmer's panel
noannotate	don't annotate stack frames (default)

? why - for testing ?

explain - ...

?

`.pcheckoff ...`
`.tcheckoff ...`
`.verify ...`

Supply checkoff information to the simulator.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures

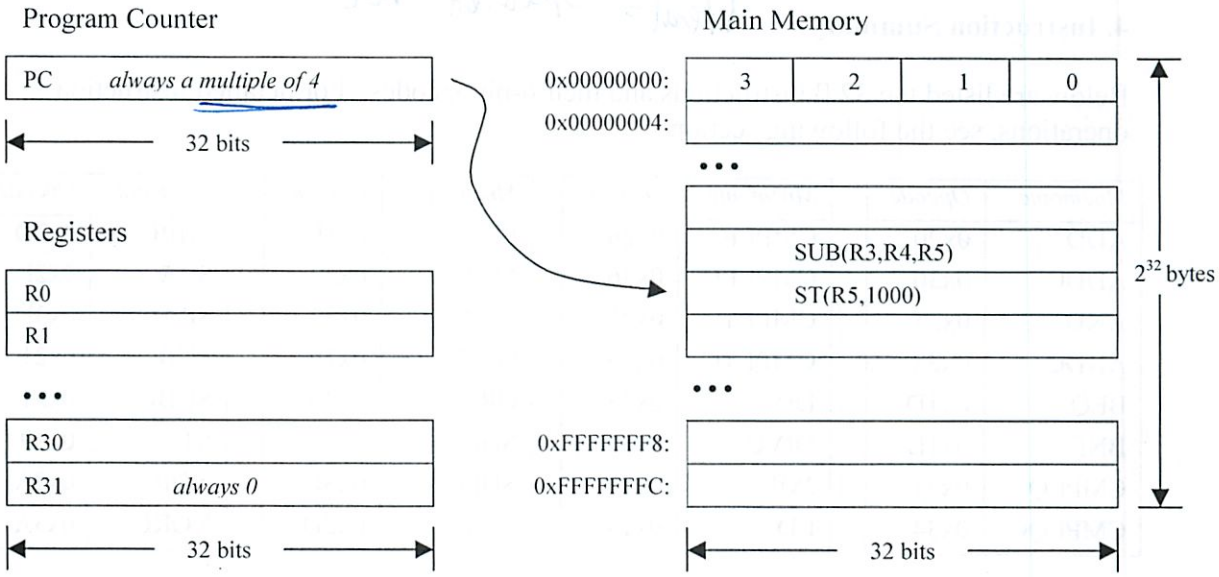
β Documentation

1. Introduction

This handout is a reference guide for the β, the RISC processor design for 6.004. This is intended to be a complete and thorough specification of the programmer-visible state and instruction set.

2. Machine Model

The β is a general-purpose 32-bit architecture: all registers are 32 bits wide and when loaded with an address can point to any location in the byte-addressed memory. When read, register 31 is always 0; when written, the new value is discarded.



3. Instruction Encoding

Each β instruction is 32 bits long. All integer manipulation is between registers, with up to two source operands (one may be a sign-extended 16-bit literal), and one destination register. Memory is referenced through load and store instructions that perform no other computation. Conditional branch instructions are separated from comparison instructions: branch instructions test the value of a register that can be the result of a previous compare instruction.

There are only two types of instruction encoding: *Without Literal* and *With Literal*. Instructions without literals include arithmetic and logical operations between two registers whose result is placed in a third register. Instructions with literals include all other operations.

Like all signed quantities on the β, an instruction's literal is represented in two's complement.

3.1 Without Literal

31 – 26	25 – 21	20 – 16	15 – 11	10 – 0
Opcode	Rc	Ra	Rb	*unused*

3.2 With Literal

31 – 26	25 – 21	20 – 16	15 – 0
Opcode	Rc	Ra	literal (two's complement)

literal = specified here

4. Instruction Summary

Below are listed the 32 β instructions and their 6-bit opcodes. For detailed instruction operations, see the following section.

Mnemonic	*Opcode*
ADD	0x20
ADDC	0x30
AND	0x28
ANDC	0x38
BEQ	0x1D
BNE	0x1E
CMPEQ	0x24
CMPEQC	0x34

Mnemonic	*Opcode*
CMPLE	0x26
CMPLEC	0x36
CMPLT	0x25
CMPLTC	0x35
DIV	0x23
DIVC	0x33
JMP	0x1B
LD	0x18

Mnemonic	*Opcode*
LDR	0x1F
MUL	0x22
MULC	0x32
OR	0x29
ORC	0x39
SHL	0x2C
SHLC	0x3C
SHR	0x2D

Mnemonic	*Opcode*
SHRC	0x3D
SRA	0x2E
SRAC	0x3E
SUB	0x21
SUBC	0x31
ST	0x19
XOR	0x2A
XORC	0x3A

5. Instruction Specifications

This section contains the specifications for the β instructions, listed alphabetically by mnemonic. No timing-dependent information is given: it is specifically assumed that there are no pathological timing interactions between instructions in this specification. Each instruction is considered atomic and is presumed to complete before the next instruction is executed. No assumptions are made about branch prediction, instruction prefetch, or memory caching.

written for people who have used other processors

usage ≠ this word order!!!!!!

5.1 ADD

Usage: ADD(Ra,Rb,Rc)

Opcode:	100000	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4 ← next instruction (normal)
Reg[Rc] ← Reg[Ra] + Reg[Rb]

The contents of register Ra are added to the contents of register Rb and the 32-bit sum is written to Rc. This instruction computes no carry or overflow information. If desired, this can be computed through explicit compare instructions.

5.2 ADDC

Usage: ADDC(Ra,literal,Rc)

Opcode:	110000	Rc	Ra	literal

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] + SEXT(literal)

The contents of register Ra are added to *literal* and the 32-bit sum is written to Rc. This instruction computes no carry or overflow information. If desired, this can be computed through explicit compare instructions.

5.3 AND

Usage: AND(Ra,Rb,Rc)

Opcode:	101000	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] & Reg[Rb]

This performs the bitwise boolean AND function between the contents of register Ra and the contents of register Rb. The result is written to register Rc.

5.4 ANDC

Usage: ANDC(Ra,*literal*,Rc)

Opcode:	111000	Rc	Ra	literal

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] & SEXT(literal)

This performs the bitwise boolean AND function between the contents of register Ra and *literal*. The result is written to register Rc.

5.5 BEQ/BF

Usage: BEQ(Ra,label,Rc)
BF(Ra,label,Rc)

Opcode:

011101	Rc	Ra	literal

Operation:
literal = ((OFFSET(label) – OFFSET(current instruction)) / 4)–1
PC ← PC + 4
EA ← PC + 4*SEXT(literal)
effetic address
TEMP ← Reg[Ra]
Reg[Rc] ← PC
if TEMP = 0 then PC ← EA

The PC of the instruction following the BEQ instruction (the updated PC) is written to register Rc. If the contents of register Ra are zero, the PC is loaded with the target address; otherwise, execution continues with the next sequential instruction.

The displacement *literal* is treated as a signed word offset. This means it is multiplied by 4 to convert it to a byte offset, sign extended to 32 bits, and added to the updated PC to form the target address.

{ registers are letters here }

5.6 BNE/BT

- no variables from 1st line here

Usage: BNE(Ra,label,Rc)
BT(Ra,label,Rc)

Opcode:

011110	Rc	Ra	literal

Operation:
literal = ((OFFSET(label) – OFFSET(current instruction)) ÷ 4)–1
PC ← PC + 4
EA ← PC + 4*SEXT(literal)
TEMP ← Reg[Ra]
Reg[Rc] ← PC
if TEMP ≠ 0 then PC ← EA

The PC of the instruction following the BNE instruction (the updated PC) is written to register Rc. If the contents of register Ra are non-zero, the PC is loaded with the target address; otherwise, execution continues with the next sequential instruction.

The displacement *literal* is treated as a signed word offset. This means it is multiplied by 4 to convert it to a byte offset, sign extended to 32 bits, and added to the updated PC to form the target address.

5.7 CMPEQ

Usage: CMPEQ(Ra,Rb,Rc)

Opcode:

100100	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
if Reg[Ra] = Reg[Rb] then Reg[Rc] ← 1 else Reg[Rc] ← 0

If the contents of register Ra are equal to the contents of register Rb, the value one is written to register Rc; otherwise zero is written to Rc.

5.8 CMPEQC

Usage: CMPEQC(Ra,literal,Rc)

Opcode:

110100	Rc	Ra	literal

Operation: PC ← PC + 4
if Reg[Ra] = SEXT(literal) then Reg[Rc] ← 1 else Reg[Rc] ← 0

If the contents of register Ra are equal to *literal*, the value one is written to register Rc; otherwise zero is written to Rc.

5.9 CMPLE

Usage: CMPLE(Ra,Rb,Rc)

Opcode:

100110	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
if Reg[Ra] ≤ Reg[Rb] then Reg[Rc] ← 1 else Reg[Rc] ← 0

If the contents of register Ra are less than or equal to the contents of register Rb, the value one is written to register Rc; otherwise zero is written to Rc.

5.10 CMPLEC

Usage: CMPLEC(Ra,literal,Rc)

Opcode:

110110	Rc	Ra	literal

Operation: PC ← PC + 4
if Reg[Ra] ≤ SEXT(literal) then Reg[Rc] ← 1 else Reg[Rc] ← 0

If the contents of register Ra are less than or eqaul to *literal*, the value one is written to register Rc; otherwise zero is written to Rc.

5.11 CMPLT

Usage: CMPLT(Ra,Rb,Rc)

Opcode:

100101	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
if Reg[Ra] < Reg[Rb] then Reg[Rc] ← 1 else Reg[Rc] ← 0

If the contents of register Ra are less than the contents of register Rb, the value one is written to register Rc; otherwise zero is written to Rc.

5.12 CMPLTC

Usage: CMPLTC(Ra,literal,Rc)

Opcode:

110101	Rc	Ra	literal

Operation: PC ← PC + 4
if Reg[Ra] < SEXT(literal) then Reg[Rc] ← 1 else Reg[Rc] ← 0

If the contents of register Ra are less than *literal*, the value one is written to register Rc; otherwise zero is written to Rc.

5.13 DIV

Usage: DIV(Ra,Rb,Rc)

Opcode:

100011	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] / Reg[Rb]

The contents of register Ra are divided by the contents of register Rb and the low-order 32 bits of the quotient are written to Rc.

5.14 DIVC

Usage: DIVC(Ra,literal,Rc)

Opcode:

110011	Rc	Ra	literal

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] / SEXT(literal)

The contents of register Ra are divided by *literal* and the low-order 32 bits of the quotient is written to Rc.

5.15 JMP

Usage: JMP(Ra,Rc)

Opcode:

011011	Rc	Ra	0000000000000000

Operation:
PC ← PC+4
EA ← Reg[Ra] & 0xFFFFFFFC
Reg[Rc] ← PC
PC ← EA

The PC of the instruction following the JMP instruction (the updated PC) is written to register Rc, then the PC is loaded with the contents of register Ra. The low two bits of Ra are masked to ensure that the target address is aligned on a 4-byte boundary. Ra and Rc may specify the same register; the target calculation using the old value is done before the assignment of the new value. The unused literal field should be filled with zeroes. Note that JMP can clear the supervisor bit (bit 31 of the PC) but not set it – see section 6.3 for details.

5.16 LD

Usage: LD(Ra,literal,Rc)

Opcode:

011000	Rc	Ra	literal

Operation:
PC ← PC+4
EA ← Reg[Ra] + SEXT(literal)
Reg[Rc] ← Mem[EA]

The effective address EA is computed by adding the contents of register Ra to the sign-extended 16-bit displacement *literal*. The location in memory specified by EA is read into register Rc.

5.17 LDR

Usage: LDR(label,Rc)

Opcode:

011111	Rc	11111	literal

Operation:
literal = ((OFFSET(label) – OFFSET(current instruction)) / 4)–1
PC ← PC + 4
EA ← PC + 4*SEXT(literal)
Reg[Rc] ← Mem[EA]

The effective address EA is computed by multiplying the sign-extended *literal* by 4 (to convert it to a byte offset) and adding it to the updated PC. The location in memory specified by EA is read into register Rc. The Ra field is ignored and should be 11111 (R31). The supervisor bit (bit 31 of the PC) is ignored (i.e., treated as zero) when computing EA.

5.18 MUL

Usage: MUL(Ra,Rb,Rc)

Opcode:	100010	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] * Reg[Rb]

The contents of register Ra are multiplied by the contents of register Rb and the low-order 32 bits of the product are written to Rc.

5.19 MULC

Usage: MULC(Ra,literal,Rc)

Opcode:	110010	Rc	Ra	literal

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] * SEXT(literal)

The contents of register Ra are multiplied by *literal* and the low-order 32 bits of the product are written to Rc.

5.20 OR

Usage: OR(Ra,Rb,Rc)

Opcode:	101001	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] | Reg[Rb]

This performs the bitwise boolean OR function between the contents of register Ra and the contents of register Rb. The result is written to register Rc.

5.21 ORC

Usage: ORC(Ra,*literal*,Rc)

Opcode:	111001	Rc	Ra	literal

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] | SEXT(literal)

This performs the bitwise boolean OR function between the contents of register Ra and *literal*. The result is written to register Rc.

5.22 SHL

Usage: SHL(Ra,Rb,Rc)

Opcode:	101100	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] << $\text{Reg[Rb]}_{4:0}$

The contents of register Ra are shifted left 0 to 31 bits as specified by the five-bit count in register Rb. The result is written to register Rc. Zeroes are propagated into the vacated bit positions.

5.23 SHLC

Usage: SHLC(Ra,*literal*,Rc)

Opcode:	111100	Rc	Ra	literal

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] << $\text{literal}_{4:0}$

The contents of register Ra are shifted left 0 to 31 bits as specified by the five-bit count in *literal*. The result is written to register Rc. Zeroes are propagated into the vacated bit positions.

5.24 SHR

Usage: SHR(Ra,Rb,Rc)

Opcode:	101101	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] >> $\text{Reg[Rb]}_{4:0}$

The contents of register Ra are shifted right 0 to 31 bits as specified by the five-bit count in register Rb. The result is written to register Rc. Zeroes are propagated into the vacated bit positions.

5.25 SHRC

Usage: SHRC(Ra,*literal*,Rc)

Opcode:	111101	Rc	Ra	literal

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] >> $\text{literal}_{4:0}$

The contents of register Ra are shifted right 0 to 31 bits as specified by the five-bit count in *literal*. The result result is written to register Rc. Zeroes are propagated into the vacated bit positions.

5.26 SRA

Usage: SRA(Ra,Rb,Rc)

Opcode:

101110	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] >> $\text{Reg[Rb]}_{4:0}$

The contents of register Ra are shifted arithmetically right 0 to 31 bits as specified by the five-bit count in register Rb. The result is written to register Rc. The sign bit (Reg[Ra]_{31}) is propagated into the vacated bit positions.

5.25 SRAC

Usage: SRAC(Ra,*literal*,Rc)

Opcode:

111110	Rc	Ra	literal

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] >> $\text{literal}_{4:0}$

The contents of register Ra are shifted arithmetically right 0 to 31 bits as specified by the five-bit count in *literal*. The result is written to register Rc. The sign bit (Reg[Ra]_{31}) is propagated into the vacated bit positions.

5.28 ST

Usage: ST(Rc,literal,Ra)

Opcode:

011001	Rc	Ra	literal

Operation: PC ← PC+4

EA ← Reg[Ra] + SEXT(literal)

Mem[EA] ← Reg[Rc]

The effective address EA is computed by adding the contents of register Ra to the sign-extended 16-bit displacement *literal*. The contents of register Rc are then written to the location in memory specified by EA.

5.29 SUB

Usage: SUB(Ra,Rb,Rc)

Opcode:

100001	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] – Reg[Rb]

The contents of register Rb are subtracted from the contents of register Ra and the 32-bit difference is written to Rc. This instruction computes no borrow or overflow information. If desired, this can be computed through explicit compare instructions.

5.30 SUBC

Usage: SUBC(Ra,literal,Rc)

Opcode:

110001	Rc	Ra	literal

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] – SEXT(literal)

The constant *literal* is subtracted from the contents of register Ra and the 32-bit difference is written to Rc. This instruction computes no borrow or overflow information. If desired, this can be computed through explicit compare instructions.

5.31 XOR

Usage: XOR(Ra,Rb,Rc)

Opcode:

101010	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] ^ Reg[Rb]

This performs the bitwise boolean XOR function between the contents of register Ra and the contents of register Rb. The result is written to register Rc.

5.32 XORC

Usage: XORC(Ra,*literal*,Rc)

Opcode:

111010	Rc	Ra	literal

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] ^ SEXT(literal)

This performs the bitwise boolean XOR function between the contents of register Ra and *literal*. The result is written to register Rc.

6. Extensions for Exception Handling

The standard β architecture described above is modified as follows to support exceptions and privileged instructions.

6.1 Exceptions

β exceptions come in three flavors: traps, faults, and interrupts.

Traps and faults are both the direct outcome of an instruction (e.g., an attempt to execute an illegal opcode) and are distinguished by the programmer's intentions. Traps are intentional and are normally used to request service from the operating system. Faults are unintentional and often signify error conditions.

Interrupts are asynchronous with respect to the instruction stream, and are usually caused by external events (e.g., a character appearing on an input device).

6.2 The XP Register

Register 30 is dedicated as the ``Exception Pointer" (XP) register. When an exception occurs, the updated PC is written to the XP. For traps and faults, this will be the PC of the instruction following the one which caused the fault; for interrupts, this will be the PC of the instruction following the one which was about to be executed when the interrupt occurred. The instruction pointed to by XP–4 has *not* been executed.

Since the XP can be overwritten at unpredictable times as the result of an interrupt, it should not be used by user-mode programs while interrupts are enabled.

6.3 Supervisor Mode

The high bit of the PC is dedicated as the "Supervisor" bit. The instruction fetch and LDR instruction ignore this bit, treating it as if it were zero. The JMP instruction is allowed to clear the Supervisor bit but not set it, and no other instructions may have any effect on it. Only exceptions cause the Supervisor bit to become set.

When the Supervisor bit is clear, the processor is said to be in "user mode". Interrupts are enabled while in user mode.

When the Supervisor bit is set, the processor is said to be in "supervisor mode". While in supervisor mode, interrupts are disabled and privileged instructions (see below) may be used. Traps and faults while in supervisor mode have implementation-defined (probably fatal) effects.

Since the JMP instruction can clear the Supervisor bit, it is possible to load the PC with a new value and enter user mode in a single atomic action. This provides a safe mechanism for returning from a trap to the Operating System, even if an interrupt is pending at the time.

6.4 Exception Handling

We should talk about in class

When an exception occurs and the processor is in user mode, the updated PC is written to the XP, the Supervisor bit is set, the PC is loaded with an implementation-defined value, and the processor begins executing instructions from that point. This value is called the "exception vector", and may depend on the kind of exception which occurred.

The only exception which must be supported by all implementations is the "reset" exception (also called the "power up" exception), which occurs immediately before any instructions are executed by the processor. The exception vector for power up is always 0. Thus, at power up time, the Supervisor bit is set, the XP is undefined, and execution begins at location 0 of memory.

6.5 Privileged Instructions

Some instructions may be available while in supervisor mode which are not available in user mode (e.g., instructions which interface directly with I/O devices). These are called "privileged instructions". These instructions always have an opcode of 0x00; otherwise, their form and semantics are implementation-defined. Attempts to use privileged instructions while in user mode will result in an illegal instruction exception.

how build this?

7. Software Conventions

This section describes our software programming conventions that supplement the basic architecture.

7.1 Reserved Registers

It is convenient to reserve a number of registers for pervasive standard uses. The hardware itself reserves R31 and R30; in addition, our software conventions reserve R29, R28, and R27.

These are summarized in the following table and are described more fully below.

Register	Symbol	Usage
R31	R31	Always zero
R30	XP	Exception pointer
R29	SP	Stack pointer
R28	LP	Linkage pointer
R27	BP	Base of frame pointer

7.2 Convenience Macros

We augment the basic β instruction set with the following macros, making it easier to express certain common operations:

Macro	Definition
BEQ(Ra, label)	BEQ(Ra, label, R31)
BF(Ra, label)	BF(Ra, label, R31)
BNE(Ra, label)	BNE(Ra, label, R31)
BT(Ra, label)	BT(Ra, label, R31)
BR(label, Rc)	BEQ(R31, label, Rc)
BR(label)	BR(label, R31)
JMP(Ra)	JMP(Ra, R31)
LD(label, Rc)	LD(R31, label, Rc)
ST(Rc, label)	ST(Rc, label, R31)
MOVE(Ra, Rc)	ADD(Ra, R31, Rc)
CMOVE(c, Rc)	ADDC(R31, c, Rc)
PUSH(Ra)	ADDC(SP, 4, SP) ST(Ra, -4, SP)

always branch

should say Move(Rc, Ra) - Add(Rc, R31, Ra)

no did this wrong

POP(Rc)	LD(SP, -4, Rc) SUBC(SP, 4, SP)
ALLOCATE(k)	ADDC(SP, 4*k, SP)
DEALLOCATE(k)	SUBC(SP, 4*k, SP)

7.3 Stack Implementation

SP is a reserved register that points to the top of the stack. The stack is an arbitrary contiguous region of memory. The contents of SP are always a multiple of 4 and each stack slot is 4 bytes. SP points to the location just beyond the topmost element on the stack. The stack grows upward in memory (i.e., towards higher addresses). Four macros are defined for manipulating the stack:

PUSH(Ra) - Push the contents of register Ra onto the stack

POP(Rc) - Pop the top element of the stack into Rc

ALLOCATE(k) - Push k words of uninitialized data onto the stack

DEALLOCATE(k) - Pop k words off of the stack and throw them away

7.4 Procedure Linkage

A procedure's arguments are passed on the stack. Specifically, when a procedure is entered, the topmost element on the stack is the first argument to the procedure; the next element on the stack is the second argument to the procedure, and so on. A procedure's return address is passed in LP, which is a register reserved for this purpose. A procedure returns its value (if any) in R0 and must leave all other registers, including the reserved registers, unaltered.

Thus, a typical call to a procedure named F looks like:

(push arg_{n-1})
...
(push arg_1)
(push arg_0)
BR (F, LP)
DEALLOCATE (n)
(use R0, which is now F(arg_0, arg_1, ... , arg_{n-1}))

7.5 Stack Frames

The preceeding section describes the rules which procedures must follow to interoperate properly. This section describes our conventional means of writing a procedure which follows those rules.

A procedure invocation requires storage for its arguments, its local variables, and any registers it needs to save and restore. All of this storage is allocated in a contiguous region of the stack called a "stack frame". Procedures "activate" stack frames on entry and "deactivate" them on exit. BP is a reserved register which points to a fixed location within the currently active stack frame. Procedures use a standard prologue and epilogue to activate and deactivate the stack frame.

The standard prologue is:

```
PUSH (LP)
PUSH (BP)
MOVE (SP, BP)
ALLOCATE (k)        | allocate space for locals
(push registers which are used by procedure)
```

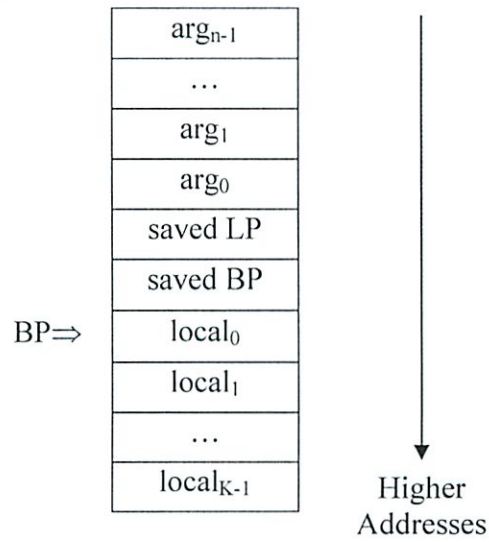
Note that either of the last two steps may be omitted if there are no local variables or if there are no registers to save.

The standard epilogue is:

```
(pop registers which are used by procedure)
MOVE (BP, SP)       | deallocate space for locals
POP (BP)
POP (LP)
JMP (LP)
```

Note that the epilogue assumes that the body of the procedure has no net effect on SP. Also note that either or both of the first two steps may be omitted if there are no registers to restore or if there are no local variables.

The standard prologue and epilogue together with the argument passing conventions imply the following layout for a stack frame:



(saved regs)

Note that BP always points to the first stack slot above the saved BP, which is the same as the first local variable (if any). It also points to the third stack slot above the first argument (if any). So within the procedure's body, its arguments and locals may be accessed via constant offsets from BP.

Summary of β Instruction Formats

Operate Class:

Register	Symbol	Usage
R31	R31	Always zero
R30	XP	Exception pointer
R29	SP	Stack pointer
R28	LP	Linkage pointer
R27	BP	Base of frame pointer

31 – 26	25 – 21	20 – 16	15 – 11	10 – 0
10xxxx	Rc	Ra	Rb	*unused*

OP(Ra,Rb,Rc): Reg[Rc] ← Reg[Ra] op Reg[Rb]

Opcodes: **ADD** (plus), **SUB** (minus), **MUL** (multiply), **DIV** (divided by)
AND (bitwise and), **OR** (bitwise or), **XOR** (bitwise exclusive or)
CMPEQ (equal), **CMPLT** (less than), **CMPLE** (less than or equal) [result = 1 if true, 0 if false]
SHL (left shift), **SHR** (right shift w/o sign extension), **SRA** (right shift w/ sign extension)

31 – 26	25 – 21	20 – 16	15 – 0
11xxxx	Rc	Ra	literal (two's complement)

OPC(Ra,literal,Rc): Reg[Rc] ← Reg[Ra] op SEXT(literal)

Opcodes: **ADDC** (plus), **SUBC** (minus), **MULC** (multiply), **DIVC** (divided by)
ANDC (bitwise and), **ORC** (bitwise or), **XORC** (bitwise exclusive or)
CMPEQC (equal), **CMPLTC** (less than), **CMPLEC** (less than or equal) [result = 1 if true, 0 if false]
SHLC (left shift), **SHRC** (right shift w/o sign extension), **SRAC** (right shift w/ sign extension)

Other:

31 – 26	25 – 21	20 – 16	15 – 0
01xxxx	Rc	Ra	literal (two's complement)

LD(Ra,literal,Rc): Reg[Rc] ← Mem[Reg[Ra] + SEXT(literal)]
ST(Rc,literal,Ra): Mem[Reg[Ra] + SEXT(literal)] ← Reg[Rc]
JMP(Ra,Rc): Reg[Rc] ← PC + 4; PC ← Reg[Ra]
BEQ/BF(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] = 0 then PC ← PC + 4 + 4*SEXT(literal)
BNE/BT(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] ≠ 0 then PC ← PC + 4 + 4*SEXT(literal)
LDR(Ra,label,Rc): Reg[Rc] ← Mem[PC + 4 + 4*SEXT(literal)]

Opcode Table: (*optional opcodes)

2:0 / *5:3*	*000*	*001*	*010*	*011*	*100*	*101*	*110*	*111*
000								
001								
010								
011	**LD**	**ST**		**JMP**		**BEQ**	**BNE**	**LDR**
100	**ADD**	**SUB**	**MUL***	**DIV***	**CMPEQ**	**CMPLT**	**CMPLE**	
101	**AND**	**OR**	**XOR**		**SHL**	**SHR**	**SRA**	
110	**ADDC**	**SUBC**	**MULC***	**DIVC***	**CMPEQC**	**CMPLTC**	**CMPLEC**	
111	**ANDC**	**ORC**	**XORC**		**SHLC**	**SHRC**	**SRAC**	

Summary of β Instruction Formats

Operate Class:

Register	Symbol	Usage
R31	R31	Always zero
R30	XP	Exception pointer
R29	SP	Stack pointer
R28	LP	Linkage pointer
R27	BP	Base of frame pointer

31 – 26	25 – 21	20 – 16	15 – 11	10 – 0
10xxxx	Rc	Ra	Rb	*unused*

OP(Ra,Rb,Rc): Reg[Rc] ← Reg[Ra] op Reg[Rb]

Opcodes: **ADD** (plus), **SUB** (minus), **MUL** (multiply), **DIV** (divided by)
AND (bitwise and), **OR** (bitwise or), **XOR** (bitwise exclusive or)
CMPEQ (equal), **CMPLT** (less than), **CMPLE** (less than or equal) [result = 1 if true, 0 if false]
SHL (left shift), **SHR** (right shift w/o sign extension), **SRA** (right shift w/ sign extension)

31 – 26	25 – 21	20 – 16	15 – 0
11xxxx	Rc	Ra	literal (two's complement)

OPC(Ra,literal,Rc): Reg[Rc] ← Reg[Ra] op SEXT(literal)

Opcodes: **ADDC** (plus), **SUBC** (minus), **MULC** (multiply), **DIVC** (divided by)
ANDC (bitwise and), **ORC** (bitwise or), **XORC** (bitwise exclusive or)
CMPEQC (equal), **CMPLTC** (less than), **CMPLEC** (less than or equal) [result = 1 if true, 0 if false]
SHLC (left shift), **SHRC** (right shift w/o sign extension), **SRAC** (right shift w/ sign extension)

Other:

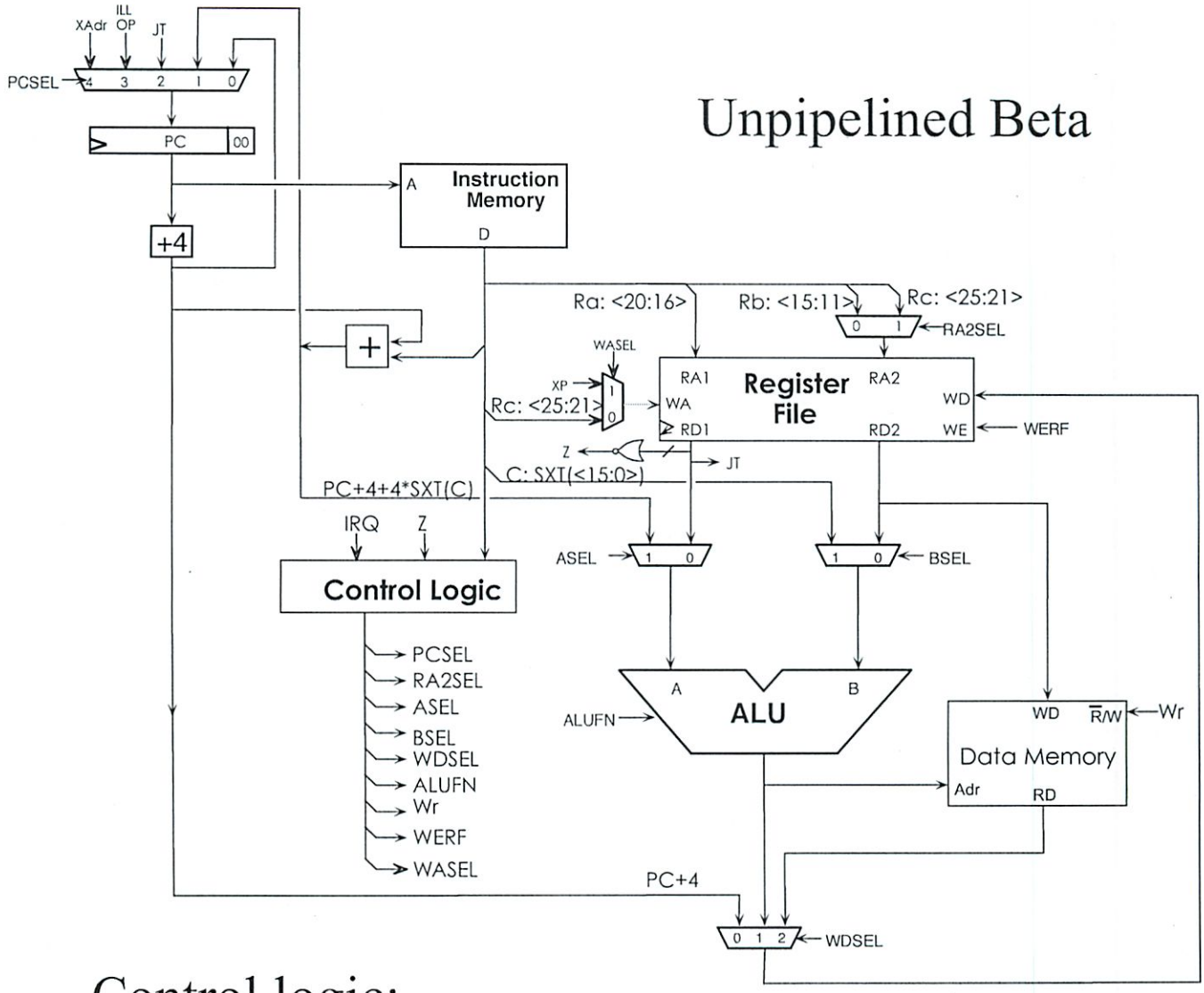
31 – 26	25 – 21	20 – 16	15 – 0
01xxxx	Rc	Ra	literal (two's complement)

LD(Ra,literal,Rc): Reg[Rc] ← Mem[Reg[Ra] + SEXT(literal)]
ST(Rc,literal,Ra): Mem[Reg[Ra] + SEXT(literal)] ← Reg[Rc]
JMP(Ra,Rc): Reg[Rc] ← PC + 4; PC ← Reg[Ra]
BEQ/BF(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] = 0 then PC ← PC + 4 + 4*SEXT(literal)
BNE/BT(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] ≠ 0 then PC ← PC + 4 + 4*SEXT(literal)
LDR(label,Rc): Reg[Rc] ← Mem[PC + 4 + 4*SEXT(literal)]

Opcode Table: (*optional opcodes)

2:0 / *5:3*	*000*	*001*	*010*	*011*	*100*	*101*	*110*	*111*
000								
001								
010								
011	**LD**	**ST**		**JMP**		**BEQ**	**BNE**	**LDR**
100	**ADD**	**SUB**	**MUL***	**DIV***	**CMPEQ**	**CMPLT**	**CMPLE**	
101	**AND**	**OR**	**XOR**		**SHL**	**SHR**	**SRA**	
110	**ADDC**	**SUBC**	**MULC***	**DIVC***	**CMPEQC**	**CMPLTC**	**CMPLEC**	
111	**ANDC**	**ORC**	**XORC**		**SHLC**	**SHRC**	**SRAC**	

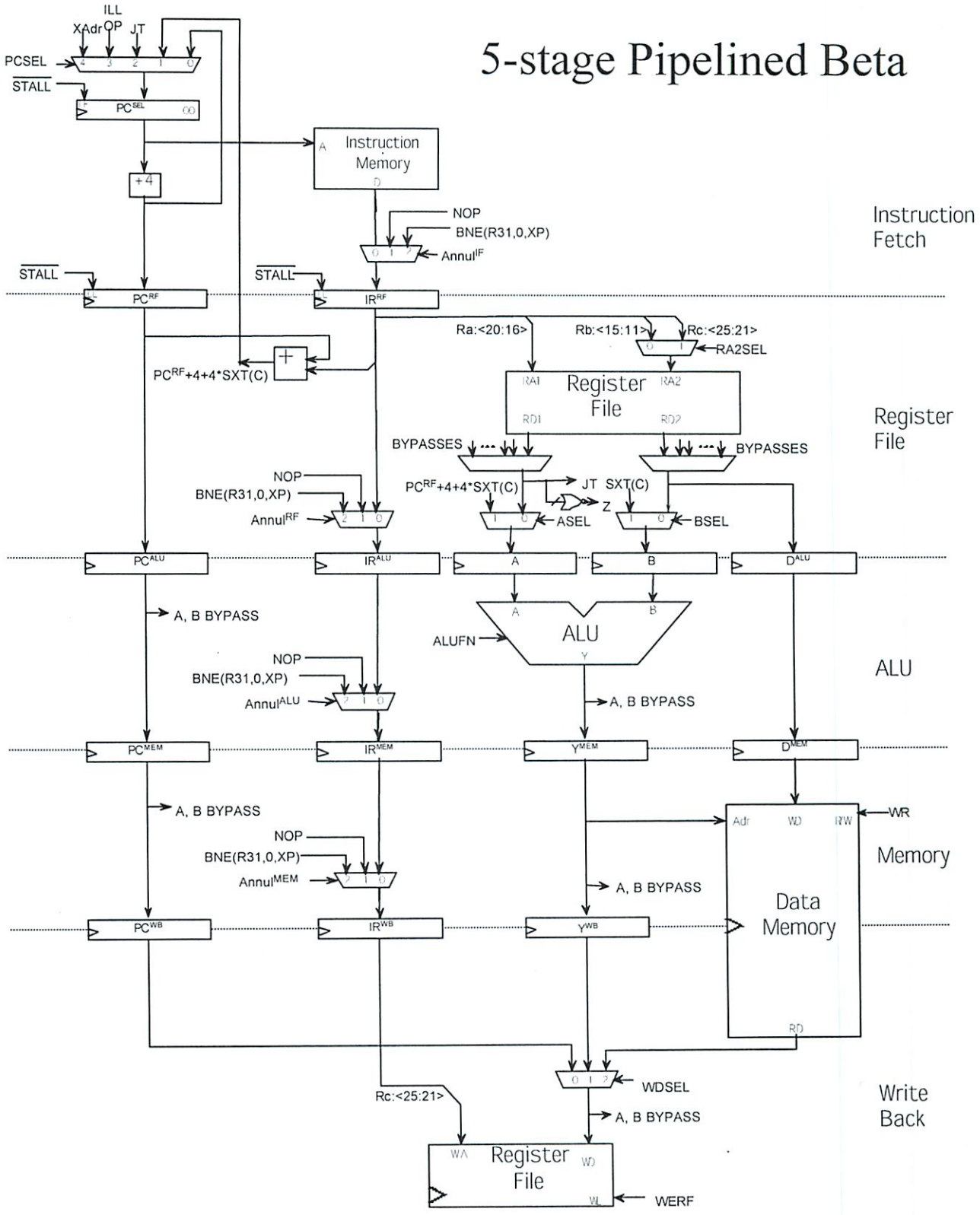
Unpipelined Beta



Control logic:

	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	ILLOP	IRQ
ALUFN	F(op)	F(op)	A+B	A+B	--	--	--	A	--	--
WERF	1	1	1	0	1	1	1	1	1	1
BSEL	0	1	1	1	--	--	--	--	--	--
WDSEL	1	1	2	--	0	0	0	2	0	0
WR	0	0	0	1	0	0	0	0	0	0
RA2SEL	0	--	--	1	--	--	--	--	--	--
PCSEL	0	0	0	0	2	Z	~Z	0	3	4
ASEL	0	0	0	0	--	--	--	1	--	--
WASEL	0	0	0	--	0	0	0	0	1	1

5-stage Pipelined Beta



Reading

10/23

WP: Bit wise ~~shift~~ operation

- at level of individual bits
- since very fast

NOT flips each bit

for 2's complement its $-x-1$

AND

both bits must be 1

```
    0101
AND 0011
    ----
    0001
```

OR

```
    0101
OR  0011
    ----
    0111
```

to set selected bits

XOR - vake a XOR itself = all 0s

Bit-wise - operate on pairs

Shifts

- Some bits shifted out or in

```
   MSB        LSB
   0 0 0 1 0 1 1 1
discard ←
     ↙ ↙ ↙ ↙ ↙ ↙ ↙
   0 0 1 0 1 1 1 0 ← 0
```

Arithmetic Left shift

②

0 0 0 1 0 1 1 1 → discard Arithmetic Right

0 0 0 0 1 0 1 1

left shift is multiplying by 2^n

- assuming no overflow

right shift is dividing by 2^n and rounding to $-\infty$

<u>Logical shifts</u> - 0s shifted in on left and right

- left: logical + arthmetic same
- right: insert 0 instead of copying sign bit
  - (remember MSB lost)
- use arthmetic for 2s complement

<u>Rotate/circular shifts</u>

- put discarded bit in
- used in crypto

<u>In C</u>

<< or >>

~~arthmetic shifts~~

logical shifts

$x = a << b$ is $x = a \cdot 2^b$ und. if overflow

$x = a >> b$ is $x = a / 2^b$

(3)

<u>Pointer</u> value that points to an address in memory

like a register, but in memory

<u>In C</u>

int *ptr

ptr points to an object of type int

Should explicitly set to null

& = address of

```
int a = 5
int *ptr = Null;
ptr = &a;
```

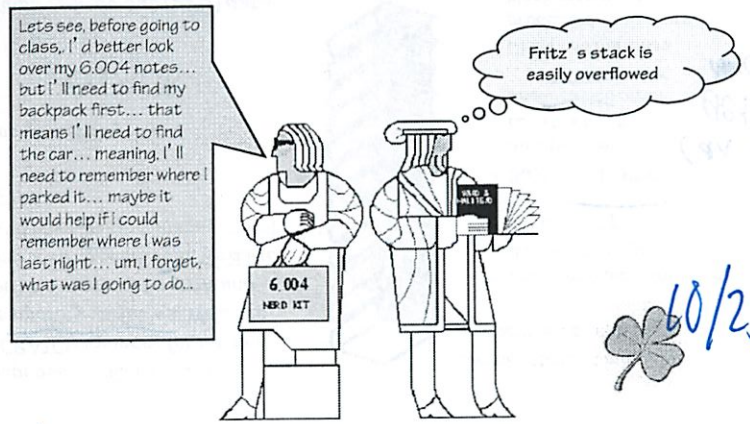
Change value of pointed-to memory location

```
*ptr = 8;
```

Skipping how to implement an array

Preview from last ~~lecture~~ semester

Stacks and Procedures

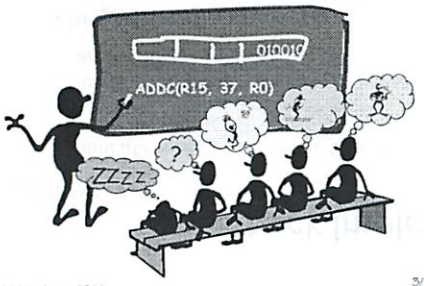


60/23

last semester Lab 4 due Today!

Where we left things last time...

```
int fact(int n)
{
    int r = 1;
    while (n>0) {
        r = r*n;
        n = n-1;
    }
    return r;
}

fact(4);
```



Procedures & Functions

- Reusable code fragments that are called as needed
- Single "named" entry point
- Parameterizable
- Local state (variables)
- Upon completion control is transferred back to caller

Procedure Linkage: First Try

```
int fact(int n)
{
    if (n>0)
        return n*fact(n-1);
    else
        return 1;
}

fact(4);
```

"Recursion" defined:
a *recursive definition* is simply a recursive definition.

fact(4) = 4*fact(3)
fact(3) = 3*fact(2)
fact(2) = 2*fact(1)
fact(1) = 1*fact(0)
fact(0) = 1 ← need end

Proposed convention:
- pass arg in R1
- pass return addr in R28
- return result in R0
- questions:
  - nargs > 1?
  - preserve regs?

Let's just use some registers. We've got plenty...

Procedure Linkage: First Try

```
int fact(int n)
{
    if (n>0)
        return n*fact(n-1);
    else
        return 1;
}

fact(3);
```

```
fact:
        CMPLEC(r1,0,r0)
        BT(r0,else)
        MOVE(r1,r2)      | save n
        SUBC(r2,1,r1)
        BR(fact,r28)
        MUL(r0,r2,r0)
        BR(rtn)
else:   CMOVE(1,r0)
rtn:    JMP(r28,r31)

main:   CMOVE(3,r1)
        BR(fact,r28)
        HALT()
```

OOPS!

Proposed convention:
- pass arg in R1
- pass return addr in R28
- return result in R0
- questions:
  - nargs > 1?
  - preserve regs?

Need: O(n) storage locations!

Revisiting Procedure's Storage Needs

Basic Overhead for Procedures/Functions:

- Arguments
  f(x,y,z) or perhaps... sin(a+b)
- Return Address when returning to caller
- Results to be passed back to caller.

In C it's the caller's job to evaluate its arguments as expressions, and pass their resulting values to the callee... Thus, a variable name is just a simple case of an expression.

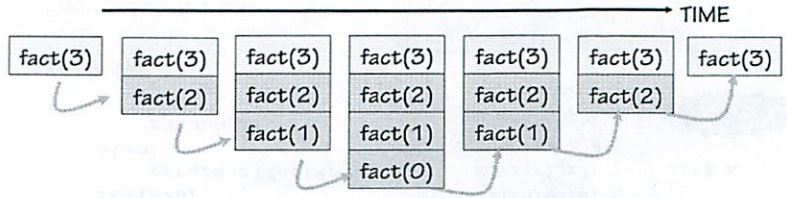
Temporary Storage:
intermediate results during expression evaluation.
(a+b)*(c+d)

Local variables:
{ int x, y;
... x ... y ...;
}

Each of these is specific to a particular *activation* of a procedure; collectively, they may be viewed as the procedure's *activation record*.

Lives of Activation Records

```
int fact(int n)
 { if (n > 0) return n*fact(n-1);
   else return 1;
 }
```



A procedure call creates a new activation record. Caller's record is preserved because we'll need it when call finally returns.

Return to previous activation record when procedure finishes, permanently discarding activation record created by call we are returning from.

Insight (ca. 1960): We need a STACK!

Suppose we allocated a SCRATCH memory for holding temporary variables. We'd like for this memory to grow and shrink as needed. And, we'd like it to have an easy management policy.

One possibility is a

STACK

A last-in-first-out (LIFO) data structure.

Some interesting properties of stacks:

- Low overhead: Allocation, deallocation by simply adjusting a pointer.
- Basic PUSH, POP discipline: strong constraint on deallocation order.
- Discipline matches procedure call/return, block entry/exit, interrupts, etc.

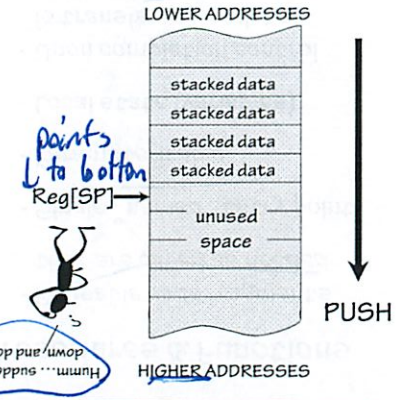
can only get it one way

Stack Implementation

CONVENTIONS:

- Dedicate a register for the Stack Pointer (SP), R29.
- Builds UP (towards higher addresses) on push
- SP points to first UNUSED location; locations below SP are allocated (protected).
- Discipline: can use stack *at any time*; but leave it as you found it!
- Reserve a block of memory well away from our program and its data

We use only *software conventions* to implement our stack (many architectures dedicate hardware)



Other possible implementations include stacks that grow "down", SP points to top of stack, etc.

Silly Convention mess

Stack Management Macros

PUSH (RX) : push Reg[x] onto stack

Reg[SP] = Reg[SP] + 4;
Mem[Reg[SP]-4] = Reg[x]

```
ADDC(R29, 4, R29)
ST(RX,-4,R29)
```

POP (RX) : pop the value on the top of the stack into Reg[x]

Reg[x] = Mem[Reg[SP]-4]
Reg[SP] = Reg[SP] - 4;

```
LD(R29, -4, RX)
ADDC(R29,-4,R29)
```

Why?

ALLOCATE (k) : reserve k WORDS of stack

Reg[SP] = Reg[SP] + 4*k

```
ADDC(R29,4*k,R29)
```

DEALLOCATE (k) : release k WORDS of stack

Reg[SP] = Reg[SP] - 4*k

```
SUBC(R29,4*k,R29)
```


Fun with Stacks

We can squirrel away variables for later. For instance, the following code fragment can be inserted anywhere within a program.

Data is popped off the stack in the opposite order that it is pushed on

```
          | Argh!!! I'm out of registers Scotty!!
          |
          PUSH(R0)                  | Frees up R0
          PUSH(R1)                  | Frees up R1
          LD(R31,dilithum_xtals, R0)
          LD(R31,seconds_til_explosion, R1)
suspense: SUBC(R1, 1, R1)
          BNE(R1, suspense, R31)
          ST(R0, warp_engines,R31)
          POP(R1)                   | Restores R1
          POP(R0)                   | Restores R0
```

AND Stacks can also be used to solve other problems...

Solving Procedure Linkage "Problems"

A reminder of our storage needs:

1) We need a way to pass arguments into procedures
2) Procedures need their own *LOCAL variables*
3) Procedures need to *call other procedures*
4) Procedures might *call themselves* (Recursion)

BUT FIRST, WE'LL COMMIT SOME MORE REGISTERS:

r27 = BP. Base ptr, points into stack to the local variables of callee
r28 = LP. Linkage ptr, return address to caller
r29 = SP. Stack ptr, points to 1st unused word

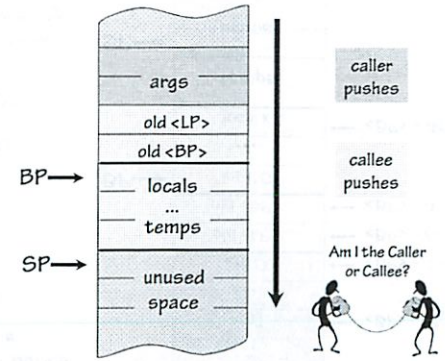
Where our local variables are stored

PLAN: CALLER puts args on stack, calls via something like
BR(CALLEE, LP)
leaving return address in LP.

"Stack frames" as activation records

The CALLEE will use the stack for all of the following storage needs:

1. saving the RETURN ADDRESS back to the caller
2. saving the CALLER's base ptr
3. Creating its own local/temp variables



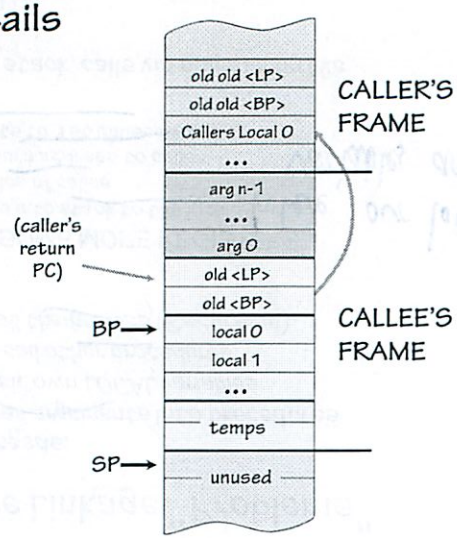
In theory it's possible to use SP to access stack frame, but offsets will change due to PUSHs and POPs. For convenience we use BP so we can use constant offsets to find, e.g., the first argument.

Stack Frame Details

The CALLER passes arguments to the CALLEE on the stack in REVERSE order

F(1,2,3,4) is translated to:

```
ADDC(R31,4,R0)
PUSH(R0)
ADDC(R31,3,R0)
PUSH(R0)
ADDC(R31,2,R0)
PUSH(R0)
ADDC(R31,1,R0)
PUSH(R0)
BEQ(R31, F, LP)
```



Why push args in REVERSE order???

So can pull off in order

Order of Arguments

Why push args onto the stack in *reverse order*?

1) It allows the BP to serve double duties when accessing the local frame

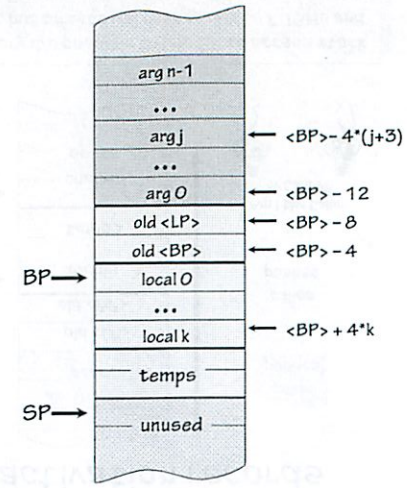
To access k^{th} local variable ($k \geq 0$)

```
LD(BP, k*4, rx)
  or
ST(rx, k*4, BP)
```

To access j^{th} argument ($j \geq 0$):

```
LD(BP, -4*(j+3), rx)
  or
ST(rx, -4*(j+3), BP)
```

2) The CALLEE can access the first few arguments without knowing how many arguments have been passed!



Procedure Linkage: The Contract

The CALLER will:

- Push args onto stack, in reverse order.
- Branch to callee, putting return address into LP.
- Remove args from stack on return.

The CALLEE will:

- Perform promised computation, leaving result in R0.
- Branch to return address.
- Leave stacked data intact, including stacked args.
- Leave regs (except R0) unchanged.

Procedure Linkage

typical "boilerplate" templates

Calling Sequence

```
        PUSH(arg_n)          | push args, last arg first
        ...
        PUSH(arg_1)
        BEQ(R31,f, LP)       | Call f.
        DEALLOCATE(n)        | Clean up!
        ...                  | (f's return value in r0)
```

Entry Sequence

```
f:      PUSH(LP)             | Save LP and BP
        PUSH(BP)             | in case we make new calls.
        MOVE(SP,BP)          | set BP=frame base
        ALLOCATE(nlocals)    | allocate locals
        (push other regs)    | preserve any regs used
```

Return Sequence

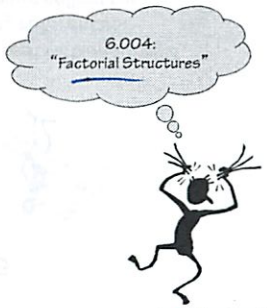
```
        (pop other regs)     | restore regs
        MOVE(val, R0)        | set return value
        MOVE(BP,SP)          | strip locals, etc
        POP(BP)              | restore CALLER's linkage
        POP(LP)              | (the return address)
        JMP(LP,R31)          | return.
```

Where's the Deallocate?

Our favorite procedure...

```
fact:  PUSH(LP)          | save linkages
       PUSH(BP)
       MOVE(SP,BP)       | new frame base
       PUSH(r1)          | preserve regs
       LD(BP,-12,r1)     | r1 ← n
       BNE(r1,big)       | if (n != 0)
       ADDC(r31,1,r0)    | else return 1;
       BR(rtn)

big:   SUBC(r1,1,r1)     | r1 ← (n-1)
       PUSH(r1)          | push arg1
       BR(fact,LP)       | fact(n-1)
       DEALLOCATE(1)     | pop arg1
       LD(BP,-12,r1)     | r0 ← n
       MUL(r1,r0,r0)     | r0 ← n*fact(n-1)

rtn:   POP(r1)           | restore regs
       MOVE(BP,SP)       | Why?
       POP(BP)           | restore links
       POP(LP)
       JMP(LP,R31)       | return.
```

```
int fact(int n)
{
  if (n != 0)
    return n*fact(n-1);
  else
    return 1;
}
```



Recursion?

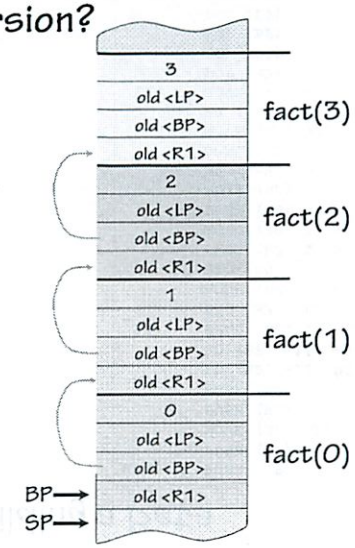
But of course!

- Frames allocated for each recursive call...
- De-allocated (in inverse order) as recursive calls return.

Debugging skill:
"stack crawling"

- Given code, stack snapshot – figure out what, where, how, who...
- Follow old <BP> links to parse frames
- Decode args, locals, return locations, etc etc etc

Particularly useful on 6.004 quizzes!

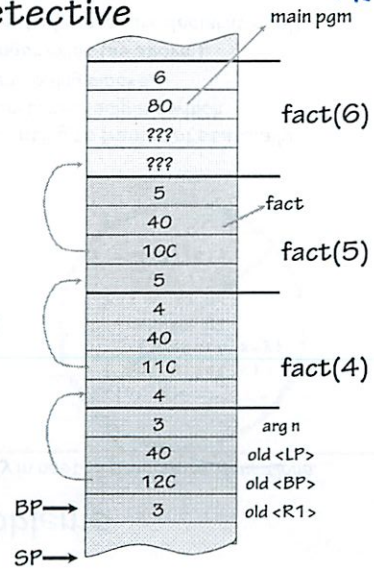


Stack Detective

Questions they ask

fact(n) is called. During the calculation, the computer is stopped with the PC at 0x40; the stack contents are shown (in hex).

- What's the argument to the *most recent* call to fact? 3
- What's the argument to the *original* call to fact? 6
- What's the location of the original calling (BR) instruction? 80 – 4 = 7C
- What instruction is about to be executed? DEALLOCATE(1)
- What value is in BP? 13C
- What value is in SP? 13C+4+4=144
- What value is in R0? fact(2) = 2
- What follows the call to fact(n)?
  another call to *fact*. Its the only program these guys have.



Man vs. Machine

Here's a C program which was fed to the C compiler*. Can you generate code as good as it did?

```
int ack(int i, int j)
{
  if (i == 0) return j+j;
  if (j == 0) return i+1;
  return ack(i-1, ack(i, j-1));
}
```

* GCC Port courtesy of Cotton Seed, Pat LoPresti, & Mitch Berger; available on Athena:

```
Athena% attach 6.004
Athena% gcc-beta -S -O2 file.c
```

Tough Problems

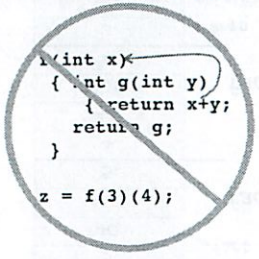
1. NON-LOCAL variable access, particularly in nested procedure definitions.

"FUNarg" problem of LISP:

```
(((lambda (x)
   (lambda(y)(+ x y)))
 3)
 4)
```

Python:

```
def f(x):
    def g(y): return x+y
    return g
z = f(3)(4)
```



Conventional solutions:

- Environments, closures.
- "static links" in stack frames, pointing to frames of statically enclosing blocks. This allows a run-time discipline which correctly accesses variables in enclosing blocks.
  BUT… enclosing block may no longer exist (as above!).

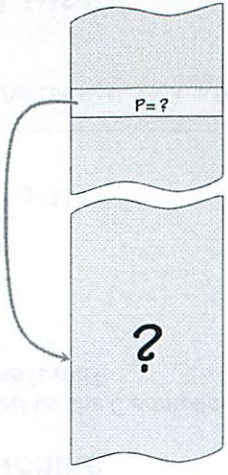
(C avoids this problem by outlawing nested procedure declarations!)

2. "Dangling References" - - -

? how about OO?

Dangling References

```
int *p; /* a pointer */

int h(x)
{
   int y = x*3;
   p = &y;
   return 37;
}

h(10);
print(*p);
```



What do we expect???

Randomness. Crashes. Smoke. Obscenities. Furious calls to Redmond, WA.

Dangling References:

different strokes...

C and C++: real tools, real dangers.

"You get what you deserve".

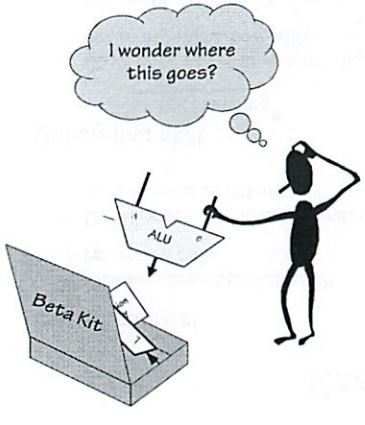
Java / Python / …: kiddie scissors only.

- No "ADDRESS OF" operator: language restrictions forbid constructs which could lead to dangling references.
- Automatic storage management: garbage collectors, reference counting: local variables allocated from a "heap" rather than a stack.

"Safety" as a language/runtime property: guarantees against stray reads, writes.

- Tension: (manual) algorithm-specific optimization opportunites vs. simple, uniform, non-optimal storage management
- Tough language/compiler problem: abstractions, compiler technology that provides simple safety yet exploits efficiency of stack allocation.

Next Time: Building a Beta




```
ack:    PUSH (LP)
        PUSH (BP)
        MOVE (SP, BP)
        PUSH (R1)
        PUSH (R2)
        LD (BP, -12, R2)
        LD (BP, -16, R0)
_L4:    SHLC (R0, 1, R1)
        BEQ (R2, _L1)
        ADDC (R2, 1, R1)
        BEQ (R0, _L1)
        SUBC (R2, 1, R1)
        SUBC (R0, 1, R0)
        PUSH (R0)
        PUSH (R2)
        BR (ack, LP)
        DEALLOCATE (2)
        MOVE (R1, R2)
        BR (_L4)
_L1:    MOVE (R1, R0)
        POP (R2)
        POP (R1)
        POP (BP)
        POP (LP)
        JMP (LP)
```

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Lab #5

Preparation: the descriptions of Beta assembly programming in lectures 11, 12 and 13 will be useful when working on this lab.

In this lab you'll have the opportunity to write your first Beta assembly language program. Your task is to write a scoring subroutine for the game of "Moo", a numeric version of Mastermind®. In Moo you try to guess the secret 4-digit number. Each guess is scored with a count of "bulls" and "cows". Each "bull" means that one of the digits in the guess matches both the value and position of a digit in the secret number. Each "cow" is a correctly guessed digit but its position in the guess doesn't match the position in the secret. Once a digit in the secret has been used to score a digit in the guess it won't be used in the scoring for other digits in the guess. The count of bulls should be determined before scoring any cows. Some examples:

← Code breaking (I think I remember playing)

Secret word: 1234	Guess: 1379	Bulls=1, Cows=1
	Guess: 4321	Bulls=0, Cows=4
	Guess: 1344	Bulls=2, Cows=1
	Guess: 1234	Bulls=4, Cows=0 (game ends!)

In addition to this handout, there are some other useful documents on the Handouts page of the course website:

BSim Documentation: describes how to use BSim, our Beta simulator with built-in assembler. Includes a brief introduction to the syntax and structure of Beta assembly language programs.

Beta Documentation: A detailed description of each instruction in the Beta instruction set. Also documents our convention for subroutine entry and exit sequences.

Summary of Instruction Formats: A one-page quick reference for Beta instructions.

Lectures 12, 13 and 14: lots of examples of Beta assembly code and compilation templates.

Moo scoring subroutine (6 points)

Your subroutine should take two arguments—the secret word and the test word—and return an integer encoding the number of bulls and cows as (16*bulls) + cows. The secret and test words contain four 4-bit digits packed into the low-order 16 bits of the word. For example, if one of the words was 1234, it would be encoded as 0x1234 where "0x" indicates hexadecimal (base 16) notation. Even though 4 bits are used to encode each digit, the words will only contain the digits 0 through 9.

hmm

You're welcome to compute the score however you'd like. In case you'd like a head start, here's one approach, written in C:

```
// Test two MOO words, report Bulls & Cows...
// Each word contains four 4-bit digits, packed into low order.
// Each digit ranges from 0 to 9.
// Returns a word whose two low-order 4-bit digits are Bulls & Cows.

int count_bull_cows(int a, int b) {
  int bulls;                    // number of bulls
  int cows;                     // number of cows
  int i, j, btemp, atry, btry, mask; //temp vars

  // Compute Bulls: check each of the four 4-bit digits in turn
  bulls = 0;
  mask = 0xF;                   // mask chooses which 4-bit digit we
check
  for (i = 0; i < 4; i = i + 1) {
    // if the 4-bit digits match, we have a bull
    if ((a & mask) == (b & mask)) {
      bulls = bulls + 1;
      // turn matching 4-bit digits to 0xF so we don't
      // count them again when computing number of cows
      a = a | mask;
      b = b | mask;
    }
    // shift mask to check next 4-bit digit
    mask = mask << 4;
  }

  // Compute Cows: check each non-0xF digit of A against all the
  // non-0xF digits of B to see if we have a match
  cows = 0;
  for (i = 0; i < 4; i = i + 1) {
    atry = a & 0xF;             // this is the next digit from A
    a = a >> 4;                 // next time around check the next digit
    if (atry != 0xF) {          // if this digit wasn't a bull
      // check the A digit against each of the four B digits
      btemp = b;                // make a copy of the B digits
      mask = 0xF;               // mask chooses which 4-bit digit we check
      for (j = 0; j < 4; j = j + 1) {
        btry = btemp & 0xF;     // this is the next digit from B
        btemp = btemp >> 4;     // next time around check the next digit
        if (btry == atry) {     // if the digits match, we've found a cow
          cows = cows + 1;
          b = b | mask;         // remember that we matched this B digit
          break;                // move on to next A digit
        }
        mask = mask << 4;
      }
    }
  }

  // encode result and return to caller
  return (bulls << 4) + cows;
}
```

There is a version of the GCC C-compiler for the Beta. Please **DO NOT** use it for this assignment – we really want you to get some experience with assembly language programs. The on-line system does keep a copy of the code used to complete the checkoff so it will be possible to check for compliance.

The test jig uses our usual convention for subroutine calls: the two arguments are pushed on the stack in reverse order (i.e., the first argument is the last one pushed on the stack) and control is transferred to the beginning of the subroutine, leaving the return address in register LP. The result should be returned in R0.

Your code should use the following template. **Be sure to include the last two lines** since they allocate space for the stack used by the test jig when calling your program.

```
| include instruction macros and test jig
.include /mit/6.004/bsim/beta.uasm
.include /mit/6.004/bsim/lab5checkoff.uasm

count_bull_cows:          | your subroutine must have this name
      PUSH(LP)            | standard subroutine entry sequence
      PUSH(BP)
      MOVE(SP, BP)
      … PUSH any registers (besides R0) used by your code …

      … your code here, leave score in R0 …

      … POP saved registers …
      MOVE(BP,SP)         | standard subroutine exit sequence
      POP(BP)
      POP(LP)
      RTN()

StackBase: LONG(.+4)      | Pointer to bottom of stack
      . = .+0x1000        | Reserve space for stack...
```

Using BSim, assemble your subroutine using the build tool. If the assembly completes without errors, BSim will bring up the display window and you can execute the test jig (which will call your subroutine) using the run ▶ or single-step ⏯ tools. The test jig will try 32 different test values and type out any error messages on the tty console at the bottom of the display window. Successful execution will result in the following printout:

```
Checking count_bull_cows:
................................
Your count_bull_cows routine passes all our tests - congrats!
```

When your subroutine passes the tests, you can complete the on-line check-in using the ✓ tool.

Implementation Notes

1. If you want to examine the execution state of the Beta at a particular point in your program, insert the assembly directive ".breakpoint" at the point where you want the simulation to halt. You can restart your program by clicking the run button, or you can click single-step button to step through your program instruction-by-instruction. You can insert as many .breakpoints in your program as you'd like.

2. If your subroutine uses registers other than R0, remember that they have to be restored to their original values before returning to the caller. The usual technique is to PUSH their value onto to the stack right after the instructions of the entry sequence and POP those values back into the registers just before starting the exit sequence.

3. Assuming you've used the subroutine entry sequence shown above, the first argument can be loaded into a register using the instruction LD(BP,-12,Rx). Similarly the second argument can be loaded using LD(BP,-16,Ry).

One way to tackle the assignment is to "hand compile" the C implementation shown above using the techniques shown in lecture:

4. Allocate a register to hold each of the variables in the C code. For example, reserve R0 and R1 for temporary values, load "a" into R2, "b" into R3, assign "bulls" to R4, etc. You'll eliminate a lot of LDs and STs by keeping your variables in registers instead of in memory locations on the stack.

explain diff - guess in later lectures

but not proper

5. See Lecture 12 for the basic techniques of converting assignment statements involving simple expressions into sequences of assembly language instructions. The instruction macro CMOVE(constant,Rx) is useful for loading small numeric constants into a register. For example, assuming that the variable "mask" has been assigned to R11, the C statement "mask = 0xF;" can be implemented in a single instruction: CMOVE(0xF,R11).

6. The CMP instructions and BEQ/BNE are useful for compiling C "if" statements. For example, assuming atry has been assigned to R7, the C fragment

```
if (atry != 0xF) { statements… }
```

can be compiled into the following instruction sequence:

```
        CMPEQC(R7,0xF,R0)       | R0 is 1 if atry==0xF, 0 otherwise
        BNE(R0,endif27)         | so branch if R0 is not zero
        … code for statements …
endif27:                        | need a unique label for each if
        …
```

7. Here's the template for compiling the a "for" statement. Note that the body of the loop is executed as long as the tests are true.

```
for (inits; tests; afters) { body… }
```

expands into the following:

```
        … code for inits …
        BR(endfor32)
for32:
        … code for body …
        … code for afters …
endfor32:
        … code for tests, Rx is 1 if tests are true …
```

```
BNE(Rx,for32)
```

8. A brief summary of C operators:

=	assignment
==	equality test (use CMPEQ, CMPEQC)
!=	inequality test (use CMPEQ, CMPEQC, reverse sense of branch)
<	less than (use CMPLT, CMPLTC)
<<	left shift (use SHL)
>>	right shift (use SRA)
&	bit-wise logical and (use AND, ANDC)
|	bit-wise logical or (use OR, ORC)
+	addition (doh!, use ADD, ADDC)

ahh

6.004 Lab5

10/23

Moo
- part of mastermind

bull - digit in correct place
Cow - digit correct, put in different place

Scoring subroutine(secret, test)
    return (16 · bulls) + cows

- 4 - 4 bit digits

1 1 1 1
8 4 2 1

8 + 4 + 2 + 1 = 15

Given C version
So basically be a human compiler for C

(2)

So it stores w/ loading variables?

Push puts on stack

└ So frees up those two registers on stack

So can look at later

BP is base point

SP is stack point

↓ I'd not know had to save each

~~So where to save each?~~

Move is

$$Move(R_a, R_c) = Add(R_a, R31, R_c)$$

which is

$$Reg[R_c] \leftarrow Reg[A] + Reg[B]$$

So not like rename

So it is saving SP as new BP

~~What other variables do~~

(3)

What variables do we have

R0 bulls

R1 cows

R2 ;

R3 ;

R4 ;

R5 btemb

R6 atry

R7 btry

R8 mask

R0 should be score

also a R1 secret int

b R2 guessed int

; So do we need to initalize?

Need read what is saved

POP(↑ into where? - 1st argument

The screte word

I never saved

a

b

keep list in code

(4)

Debugging this will be a pain in the a*

Save ()

└ add (

(This could use a nice autocomplete IDE :))

How to do a for loop?

└ see pg 4

R a is from

R c is to

Now I see why nice to pre allocate variables

FOR is actually simple...

Now need to do if

Need to make sure I fully understand branching

First need to do ~~branching~~ testing

a ~~&~~ ~~and~~ mask is what?

- bitwise logical AND

AND

put temp registers

(5)

R11 first part
R12 2nd
R13 if =

Branch if false
└otherwise continue

Wait CMPEQ

?? what is going on w/ mask??
- I don't really care as compiler

| = Bitwise logocial or

digit not equal = digit not equal =.
STILL

getting faster at assembly coding
what is BR?

BEQ (R31, label, RC)
↑BF

R31

So if label is false
go there
don't save

(6)

So set 1 for end for 32 when its 1

? So is the code they gave is wrong?

I'm doing it kinda differently

Remember it runs into next section automatically

CMPLEC

$i \leq 3$

since

No

CMLTC Syntatically better

No when this is true (1) branch back

Oh BR is always false

Oh to do test first

got it now

Done bills

— but untested

7

Cows

& = AND

0xF is what ~~16~~?
15?
1111

SQA
↑ right

Indentations don't matter

break goes where
└ terminates for loop its immediately inside

~~Cmh~~
End
why shift balls?

1st draft done
Now test
- first a bunch of typos
└ so lucky to have all these test tools

(8)

Ok runs now

(X) Fails

Lok gir this will be very annoying to debug

It also runs the debug code when executing...

This will be annoying...

It always returns 0

Read notes

1. Can put in breakpoint
2. Must save other registers to stack before and after
3. First argument LD(BP,-12,Rx)

? Why not Pop()?

↳ is a macro

↓ the parameter

LD(R29, -4, Rx)

ADDC(R29, -4, R20)

?? So this storing BP, SP means we are doing more complex stuff on stack?

(9)

I could also do rest in lab
But try break point stuff

Why are registers not cleared at
had ADD (R14, R31, R31)
L should be R31 + R3 → R14
0 + 0 → R14

Ok R1, 2 should not be
But R11, and 12
I have no clue why its not clearing
Go to lab hrs

6.004 On-line: Questions for Lab 5

When you're done remember to save your work by clicking on the "Save" button at the bottom of the page. You can check if your answers are correct by clicking on the "Check" button.

When entering numeric values in the answer fields, you can use integers (1000), floating-point numbers (1000.0), scientific notation (1e3), or JSim numeric scale factors (1K).

Problem 1. For each of the Beta instruction sequences shown below, indicate the values of the specified registers after the sequence has been executed by an unpipelined Beta. Consider each sequence separately and assume that execution begins at location 0 and halts when the HALT() instruction is about to be executed. Also assume that all registers have been initialized to 0 before execution begins.

You will find it helpful to read the note above for information on how to enter hex constants (useful for entering addresses and values). Remember that even though the Beta reads and writes 32-bit words from memory, all addresses are *byte addresses*, i.e., the addresses of successive words in memory differ by 4.

You can find detailed descriptions of each Beta instruction in the "Beta Documentation" handout distributed in lecture and available on-line.

A.

```
. = 0
AND(r31, r31, r0)
CMPEQ(r31, r31, r1)
ADD(r1, r1, r2)
OR(r2, r1, r3)
SHL(r1, r2, r4)
HALT()
```

0 - does nothing saving to R31

0

← R1 = R1 + R2 0 + 0 = 0

R2 = R1 | R3 0 | 0 = 0

R1 = R2 << R4 by 0 << by 0 = 0

Value left in R0?: 0

Value left in R1?: 0

Value left in R2?: 0

Value left in R3?: 0

→ other sheet

Address of memory location containing OR instruction?:

Now I know order was wrong...

B.

```
. = 0
ADDC(r31, N, r0)
LD(r0, 8, r1)
SRAC(r1, 4, r2)
ST(r2, 4, r0)
```

r0 = r31 + N 0 + N = N

r1 = Mem(r0) + 8 N + 8 = N + 8

Oh N part → see sheet

```
    HALT()

    . = 0x2000
N:  LONG(0x12345678)
    LONG(0xDEADBEEF)
    LONG(0xEDEDEDED)
    LONG(0x00000004)
```

Value left in R0?:

Value left in R1?:

Value left in R2?:

Address of memory location written by ST?:

Value found in memory location with address 0?:

C.

```
    . = 0
    LD(r31, X, r0)
    CMPLE(r0, r31, r1)
    BNE(r1, L1, r31)
    ADDC(r31, 17, r2)
    BEQ(r31, L2, r31)
L1: SUB(r31, r0, r2)
L2: XORC(r2, 0xFFFF, r2)    | be careful here!
    HALT()

    . = 0x1CE8
X:  LONG(0x87654321)
```

Value left in R0?:

Value left in R1?:

Value left in R2?:

Value uasm assigns to L1?:

Value found in memory location with address 8?:

D.

```
    . = 0
```

```
    ADDC(r31, 0, r0)
    LD(r31, N, r1)
    BEQ(r31, L3, r31)
L1: ANDC(r1, 1, r2)
    BEQ(r2, L2, r31)
    ADDC(r0, 1, r0)
L2: SHRC(r1, 1, r1)
L3: BNE(r1, L1, r31)
    HALT()

    . = 0x2468
N:  LONG(0x8F2E3D4C)
```

Value left in R0?:

Value left in R1?:

Number of times instruction labeled L2 is executed?:

Suppose that the instructions above were relocated so that the first instruction were at location 0x100 instead of location 0. Assuming we then started execution at location 0x100 and we wanted the instructions to perform the same computation, which instruction encodings should be changed when relocating the program?

Instructions that need to be changed?:

--select answer--

E.

```
    . = 0
    BEQ(r31, L1, r0)
    ADDC(r0, 0, r0)
L1: LD(r0, 0, r1)
    HALT()
```

Value left in R0?:

Value left in R1?:

Check Save

source: on_line_questions.py, lab5questions.xdoc

Lab 5
Questions

10/23

1. Isn't everything 0?

Since nothing is ever set otherwise

Can set it up in simulator

Sim: The CMPEQ puts a 1 in R1

CMPEQ

If R[a] = R[b]

Then R[c] = 1

Else R[c] = 0

~~Oh Oops forgot it~~

So it says (3) = R 1

Which are both 0 - so true, it writes 1 to R3

Which does nothing

But how does R1 get set??

(2)

Ohhhhhh the function's usage is

CMPEQ (Ra, Rb, Rc)

For the whole assignment I was using order as in the word.

Gccc!!!!!!!!

Now need to ~~chage~~ change that on whole p^set

So lets finish qv first

CMPEQ (R31, R31, R1)

So 0==0 so 1 in R1 ✓

Next add 1+1 = R2 = 2 ✓

Now or 2 | 1 = ?

well bitwise

0010
0001 is 0011 = 3 ✓

(3)

Now where in memory is Or instruction?

So first line is 000
2nd line 004
3rd 008
4th (or) 00C ✓

B) Ok forgot N part
I was wondering how that worked - try
So 2nd part actually ins lst
~~Setar~~ What does . do again?
└ next byte address of memory
So starting at 0x200
Write a Long 0x12345678
Simulator Mem[0x2000] = 0x12345678
[0x2004] = 0x DEADBEFF
↓ etc

(4)

Then what does N; mean

"X;" is "X = ."

So ~~the~~ assembler variable N is . which is 0x2000

So ADDC (r31, N, r0)

means (opposite order of what I thought)

r0 = 0x2000

Next LD(R0, 8, R1)

Load 0x2008 into R1

which is 3rd Long statement

0xEDEDEDED = R1

pointers :)

Next SRAC(R1, 4, R2)

R1 >> 4

So do a aritmetic right shift by 4 bits

D = ~~14~~ 13 1 1 0 1

8 4 2 1

E = 14 1 1 1 0

(5)

1110 1101 1110 1101 1110 1101 1110 1101

shift → 4 bits

arthmetic so MSB is copied in wp

(but we are little endian)

Stack overflow shifts still work same

So

$\underbrace{1111}_{F}$ E D E D E D E ✓

ST(r2, 4, r0)

So r2 is written into r0 + 4

0x2000

So 0x2004 ✓

So what is in memory at 0

Sim: C01 F2000

WTF?

Oh the first instruction ADDC(R31, 8192, R0) ✓

↳ I must be entering it wrong

↳ ask in OH

6

c) LD(r31, x, r0)

So load x into R0

X is 0xKE8

So R0 = 0x87654321

CMPLE

r0 ≤ R31 R1

So clearly no 0 1

oh perhaps since 2s Complement?
-but first digit is 0 ??

BWE(R1, L1, r31)

So looking at R1 - is ~~0~~ 1

So ~~not~~ going ~~anywhere~~ to L1

ADD((r31, 17, r2)

Sub r31 -r0 save r2
So -r0 is there
then XOR(on next pg

r2 = 17

BEQ (r31, L2, R31)

So checking if R31 is zero - always true
jump L2

(7)

XORC (R2, 0xFEFF, R2)

Boolean bitwise XOR

17 → 10001

1111 1111 1111 1111
F

XOR - only one can be true

1111 1111 1110 1110
F F E E

Halt()

-17 is 101111 says LA

FF 1101 0000
13 0
C

Sim says 87654320
I think I messed up somewhere else..

(8)

Value assigned to L1

6th item

but add 4 each time

1st 000
2nd 004
3rd 008
4 00C
5 010
6 014

Value in memory location 8

So 3rd line

BNE instruction

0x 7BF10002

Another to ask about

9

D]

=0

ADDC(r31, 0, r0)

r0 = 0

LD(r31, N, r1)

r1 = 0F2E3D4C

BEQ(r31, L3, r31)

always happens → L3

BNE(r1, L1, r31)

go again → L1

ADDC(r1, 1, r2)

r2 = 0F2E3D4D

BEQ(r2, L2, r31)

never branches

ADDC(r0, 1, r0)

r0 = 1

(10)

SHRL(r1, 1, r1)

r1 >> 1

zeros put in (Logical shift)

~~0000~~

~~0001~~ wrong value

~~0010~~ wrong way

so 3 now = ~~r0~~ r1

BNE(r1, L1, r31)

r1 good so transfers

∞ loop!

No sim does not show that

R0 = 0x11 = 17

R1 = 0

Oh step by step it repeats 17 times

I did R0 not R1 and did wrong way

It will go to 0 eventually

L2 executed 32 times (counted sim)

(11)

What changes if start 0x100
↳ none in our code (✓)
but assembler will act differently

E)

$=0$

BEQ(r31, L1, r0)

~~if r31 = L1~~

if r31 is 0 → go L1
↳ (always)

LD(r0, 0, r1)

So loads location at r0 to memory call R1

Both r0, r1 = 0? (✗)

No remember instructions are at mem location 0!

↓ means programe counter...

Simi R0 0x80000004 ← PC of instruction following
R1 0x C000 0000

Actually we saved our current pointer in r0
Load that position in

(12)

R0 is just 4

R1 is C0000000 ← the ADDC instruction ?? ✓

Now my two memory location questions

(oh my original answer worked on B ...

(How would we get that w/o sim ??)

c) Its the BNE (R1, L1)

BNE (R1, L1, R31)

So

opcode	Rc=R31	Ra=R1	literal
0111, 10			
	11111	00001	0000 0000
7	B	1	0 0
	E		0000 0110
			0 6

which matches what computer gave

Emailing - is it wrong?

Sim gives 0002 as literal

0014 is PC for L1

Emailed in

6.004
Lab 5

10/23
Later

So I need to flip all of the statements
actually fairly easy to do
just take first and put last

Try
Program halts early?
Had wrong jump
Fixed now returns a value - but wrong
So need to debug...

bulls 2

Oh so it gives in + out
Should be ~~40h~~ x 40 - getting 0x2
1000000 10

So what is bit shift - ? not happening?
Oh need SHLC
Now getting 20s

(2)

So I am missing stuff in program
Only clicking on 1 and 4
Check mask
- moving correctly
(Debugging is actually kinda easy)
Is bitwise and right?
Oh guess is D4F
Oh reading it wrong
- See their suggestion for loading arguments
  - at diff pts on stack
- Since other stuff pushed on in meantime
  - the LP, BP
- Ok - I understand now
(J) Passes test 1
Fails when Guess 0000 Actual 5678
? is it reading stack correctly?
  - should be 0 according to this

(3)

It says x 1000
vs 5000 should return 30

I don't see those values in test
? Not saving other registers?

Yeah need to do that
It pulls from BP - so don't care about

- this pulling / poping seems silly ...
a better way?
diff subroutines have diff registers?

(✓) Getting a bunch more tests
But a is always clear at end
- we do change it so don't when cows

Error on 1303
2737

- Cows work else where
- Now diff error 8331 got 12
6~~8~~368 want 11

(4)

Now 2683
2065

returned 10
wanted 4

Is it randomizing test cases?

It misses some cows

Why is a 0 though

Is bulls if is correct

Now A seems to be changing

No is shifted

Oh forgot SRAC on inner loop

(V) A bunch more test cases
- feels like 20
(I wish you could backup)

0x 3231
3303

returned 12
wanted 12

that is edge case - how to score?

(5)

0x 1463 returned 3
0x 8344 wanted 2

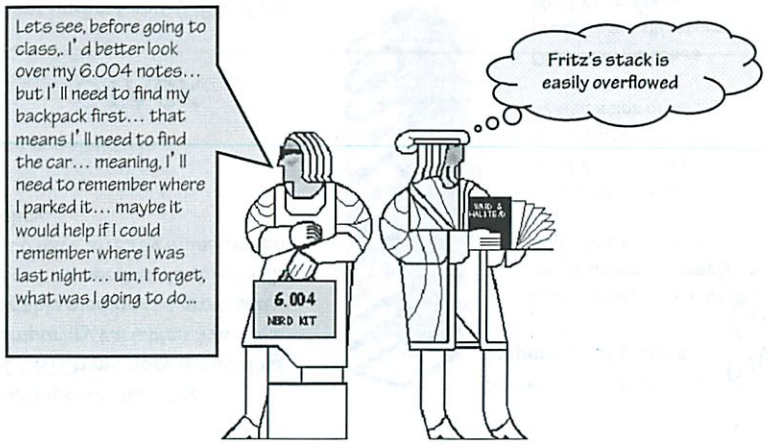
0x5975 returned 2
0x 2737 wanted 1

So matches twice
But I copied their algorithm....
Can it ∞ loop sometime
- inner loop?
? Should be BF?
└ no should be true

Oh break should always exit?
(✓) Passes test

10/25

Stacks and Procedures

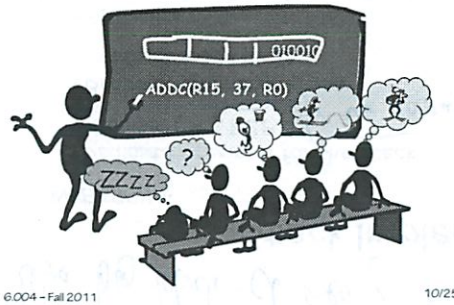


Lab 5 due Thursday!

Code changed
- redownload Bsim

Where we left things last time...

```
int fact(int n)
{
    int r = 1;
    while (n>0) {
        r = r*n;
        n = n-1;
    }
    return r;
}

fact(4);
```



Procedures & Functions

- Reusable code fragments that are called as needed
- Single "named" entry point
- Parameterizable
- Local state (variables)
- Upon completion control is transferred back to caller

packaged up

Procedure Linkage: First Try

```
int fact(int n)
{
    if (n>0)
        return n*fact(n-1);
    else
        return 1;
}

fact(4);
```

"Recursion" defined:
a *recursive definition* is simply a recursive definition.

fact(4) = 4*fact(3)
fact(3) = 3*fact(2)
fact(2) = 2*fact(1)
fact(1) = 1*fact(0)
fact(0) = 1

can do and be well defined

Now need Conventions

Proposed convention:
- pass arg in R1
- pass return addr in R28
- return result in R0
- questions:
  - nargs > 1?
  - preserve regs?

Let's just use some registers. We've got plenty...

base case and builder

go he can come back

Procedure Linkage: First Try

```
int fact(int n)
{
    if (n>0)
        return n*fact(n-1);
    else
        return 1;
}

fact(3);
```

Proposed convention:
- pass arg in R1
- pass return addr in R28
- return result in R0
- questions:
  - nargs > 1?
  - preserve regs?

```
fact:
        CMPLEC(r1,0,r0)
        BT(r0,else)
        MOVE(r1,r2)      | save n
        SUBC(r2,1,r1)
        BR(fact,r28)
        MUL(r0,r2,r0)
        BR(rtn)
else:   CMOVE(1,r0)
rtn:    JMP(r28,r31)

main:   CMOVE(3,r1)
        BR(fact,r28)
        HALT()
```

come back

OOPS!

go to r28 is location

but if call factorial recursively will overwrite previous r28

Need: O(n) storage locations!

Any nested procedure calls need O(n) registers

10/25

Rethink all storage needs

Revisiting Procedure's Storage Needs

Basic Overhead for Procedures/Functions:

- Arguments
  f(x,y,z) or perhaps... sin(a+b)
- Return Address when returning to caller
- Results to be passed back to caller.

In C it's the caller's job to evaluate its arguments as expressions, and pass their resulting *values* to the callee... Thus, a variable name is just a simple case of an expression.

Temporary Storage:
intermediate results during expression evaluation.
(a+b)*(c+d)

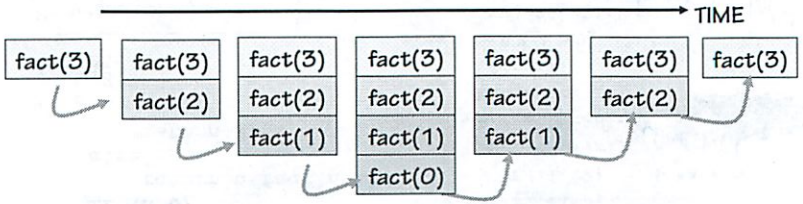
Local variables:

```
{ int x, y;
  ... x ... y ...;
}
```

Each of these is specific to a particular *activation* of a procedure; collectively, they may be viewed as the procedure's *activation record*.

Lives of Activation Records

```
int fact(int n)
 { if (n > 0) return n*fact(n-1);
   else return 1;
 }
```



A procedure call creates a new activation record. Caller's record is preserved because we'll need it when call finally returns.

Return to previous activation record when procedure finishes, permanently discarding activation record created by call we are returning from.

Very basic idea in all modern computers

Insight (ca. 1960): We need a STACK!

Suppose we allocated a SCRATCH memory for holding temporary variables. We'd like for this memory to grow and shrink as needed. And, we'd like it to have an easy management policy.

One possibility is a

STACK

A last-in-first-out (LIFO) data structure.

Some interesting properties of stacks:

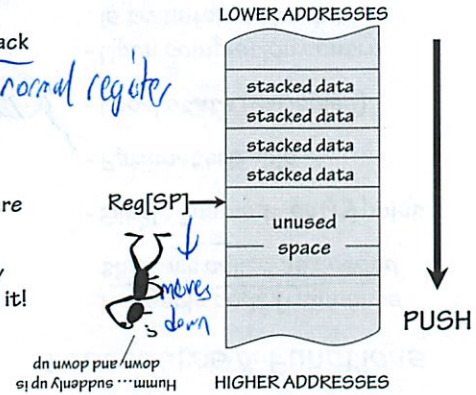
- Low overhead: Allocation, deallocation by simply adjusting a pointer.
- Basic PUSH, POP discipline: strong constraint on deallocation order.
- Discipline matches procedure call/return, block entry/exit, interrupts, etc.

Take big block of memory for stack

Stack Implementation

CONVENTIONS:

- Dedicate a register for the Stack Pointer (SP), R29. for the normal register
- Builds UP (towards higher addresses) on push
- SP points to first UNUSED location; locations below SP are allocated (protected).
- Discipline: can use stack *at any time*; but leave it as you found it!
- Reserve a block of memory well away from our program and its data



We use only *software conventions* to implement our stack (many architectures dedicate hardware)

Other possible implementations include stacks that grow "down", SP points to top of stack, etc.

Stack Management Macros

PUSH (RX) : push Reg[x] onto stack

Reg[SP] = Reg[SP] + 4;
Mem[Reg[SP]-4] = Reg[x]

```
ADDC(R29, 4, R29)
ST(RX,-4,R29)
```

← where stack used to point

POP (RX) : pop the value on the top of the stack into Reg[x]

Reg[x] = Mem[Reg[SP]-4]
Reg[SP] = Reg[SP] - 4;

```
LD(R29, -4, RX)
ADDC(R29,-4,R29)
```

Why?

Order very important
– several weeks till ans

ALLOCATE (k) : reserve k WORDS of stack

Reg[SP] = Reg[SP] + 4*k

```
ADDC(R29,4*k,R29)
```

DEALLOCATE (k) : release k WORDS of stack

Reg[SP] = Reg[SP] - 4*k

```
SUBC(R29,4*k,R29)
```

as a procedure can push and pop but you should leave stack as you found it

Fun with Stacks

We can squirrel away variables for later. For instance, the following code fragment can be inserted anywhere within a program.

Data is popped off the stack in the opposite order that it is pushed on

```
        | Argh!!! I'm out of registers Scotty!!
        |
        PUSH(R0)                    | Frees up R0
        PUSH(R1)                    | Frees up R1
        LD(R31,dilithum_xtals, R0)
        LD(R31,seconds_til_explosion, R1)
suspense: SUBC(R1, 1, R1)
        BNE(R1, suspense, R31)
        ST(R0, warp_engines,R31)
        POP(R1)                     | Restores R1
        POP(R0)                     | Restores R0
```

AND Stacks can also be used to solve other problems...

Solving Procedure Linkage "Problems"

A reminder of our storage needs:

1) We need a way to *pass arguments* into procedures
2) Procedures need their own *LOCAL variables*
3) Procedures need to *call other procedures*
4) Procedures might *call themselves* (Recursion)

BUT FIRST, WE'LL COMMIT SOME MORE REGISTERS:

r27 = BP. Base ptr, points into stack to the local variables of callee — base of frame
r28 = LP. Linkage ptr, return address to caller — return address
r29 = SP. Stack ptr, points to 1st unused word

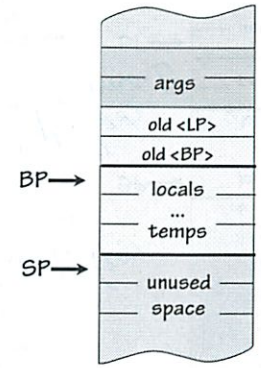
PLAN: CALLER puts args on stack, calls via something like
BR(CALLEE, LP)
leaving return address in LP.

we need to manage this ourselves

"Stack frames" as activation records

The CALLEE will use the stack for all of the following storage needs:

1. saving the RETURN ADDRESS back to the caller
2. saving the CALLER's base ptr
3. Creating its own local/temp variables



In theory it's possible to use SP to access stack frame, but offsets will change due to PUSHs and POPs. For convenience we use BP so we can use constant offsets to find, e.g., the first argument.

less readable but lots of registers to use

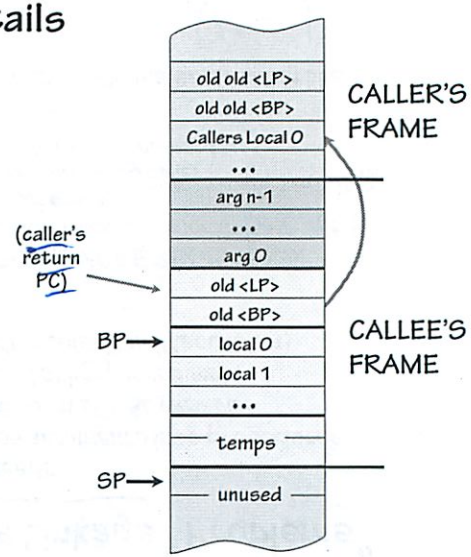
Stack Frame Details

The CALLER passes arguments to the CALLEE on the stack in REVERSE order

F(1,2,3,4) is translated to:

```
ADDC(R31,4,R0)
PUSH(R0)
ADDC(R31,3,R0)
PUSH(R0)
ADDC(R31,2,R0)
PUSH(R0)
ADDC(R31,1,R0)
PUSH(R0)
BEQ(R31, F, LP)
```

Why push args in REVERSE order???



Order of Arguments

Why push args onto the stack in *reverse* order?

1) It allows the BP to serve double duties when accessing the local frame

To access k^{th} local variable ($k \geq 0$)

```
LD(BP, k*4, rx)
  or
ST(rx, k*4, BP)
```

Offsets from 4

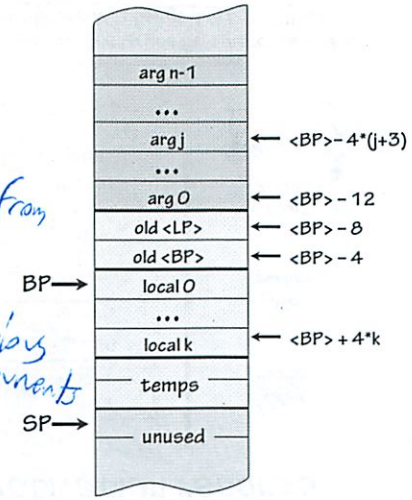
To access j^{th} argument ($j \geq 0$):

```
LD(BP, -4*(j+3), rx)
  or
ST(rx, -4*(j+3), BP)
```

previous arguments

2) The CALLEE can access the first few arguments without knowing how many arguments have been passed!

Don't want to depend on # arguments passed



Procedure Linkage: The Contract

The CALLER will:

- Push args onto stack, in reverse order.
- Branch to callee, putting return address into LP.
- Remove args from stack on return.

this is ARGS

The CALLEE will:

- Perform promised computation, leaving result in R0.
- Branch to return address.
- Leave stacked data intact, including stacked args.
- Leave regs (except R0) unchanged.

no it must. Must restore

in P-Set Callee did storing!!

Procedure Linkage

typical "boilerplate" templates

Calling Sequence

```
PUSH(arg_n)                 | push args, last arg first
...
PUSH(arg_1)
BEQ(R31,f, LP)              | Call f.
DEALLOCATE(n)               | Clean up!
...                         | (f's return value in r0)
```

The stuff we put on stack

Entry Sequence

```
f:  PUSH(LP)                | Save LP and BP
    PUSH(BP)                | in case we make new calls.
    MOVE(SP,BP)             | set BP=frame base
    ALLOCATE(nlocals)       | allocate locals
    (push other regs)       | preserve any regs used
```

Oh that's why

did allocate

Where's the Deallocate?

Return Sequence

```
(pop other regs)            | restore regs
MOVE(val, R0)               | set return value
MOVE(BP,SP)                 | strip locals, etc
POP(BP)                     | restore CALLER's linkage
POP(LP)                     | (the return address)
JMP(LP,R31)                 | return.
```

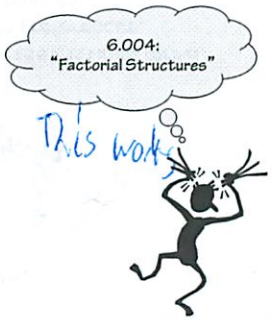
but Move(BP,SP) does built deallocate

Our favorite procedure… Factorial!

```
fact:  PUSH(LP)          | save linkages
       PUSH(BP)
       MOVE(SP,BP)       | new frame base
       PUSH(r1)          | preserve regs
       LD(BP,-12,r1)     | r1 ← n
       BNE(r1,big)       | if (n != 0)
       ADDC(r31,1,r0)    | else return 1;
       BR(rtn)

big:   SUBC(r1,1,r1)     | r1 ← (n-1)
       PUSH(r1)          | push arg1
       BR(fact,LP)       | fact(n-1)
       DEALLOCATE(1)     | pop arg1
       LD(BP,-12,r1)     | r0 ← n
       MUL(r1,r0,r0)     | r0 ← n*fact(n-1)

rtn:   POP(r1)           | restore regs
       MOVE(BP,SP)       | Why?
       POP(BP)           | restore links
       POP(LP)
       JMP(LP,R31)       | return.
```

calling seq

```
int fact(int n)
{
  if (n != 0)
    return n*fact(n-1);
  else
    return 1;
}
```



This works

Recursion?

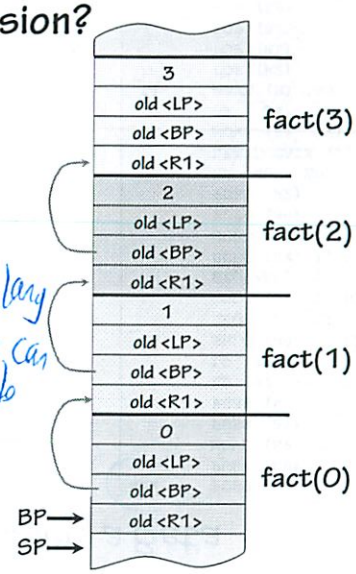
But of course!

- Frames allocated for each recursive call…
- De-allocated (in inverse order) as recursive calls return.

Debugging skill: "stack crawling" assembly lang programers can do

- Given code, stack snapshot – figure out what, where, how, who…
- Follow old <BP> links to parse frames
- Decode args, locals, return locations, etc etc etc

Particularly useful on 6.004 quizzes!



can chase from what happened

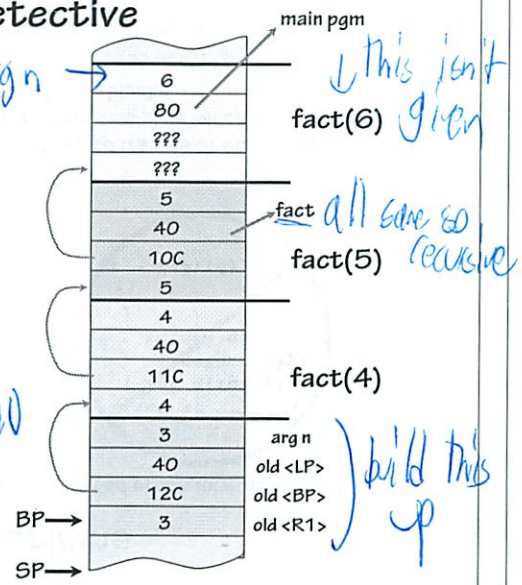
need code + stack dump

Stack Detective

fact(n) is called. During the calculation, the computer is stopped with the PC at 0x40; the stack contents are shown (in hex).

- What's the argument to the *most recent* call to fact? 3
- What's the argument to the *original* call to fact? 6
- What's the location of the original calling (BR) instruction? 80 – 4 = 7C
- What instruction is about to be executed? DEALLOCATE(1)
- What value is in BP? 13C – adding 10
- What value is in SP? 13C+4+4=144
- What value is in R0? fact(2) = 2
- What follows the call to fact(n)? another call to fact. Its the only program these guys have.

Joke!



↓ this isn't given

all same so recursive

build this up

Man vs. Machine

Here's a C program which was fed to the C compiler*. Can you generate code as good as it did?

```
int ack(int i, int j)
{
  if (i == 0) return j+j;
  if (j == 0) return i+1;
  return ack(i-1, ack(i, j-1));
}
```

* GCC Port courtesy of Cotton Seed, Pat LoPresti, & Mitch Berger; available on Athena:

```
Athena% attach 6.004
Athena% gcc-beta -S -O2 file.c
```

Tough Problems

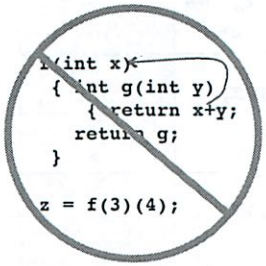
1. NON-LOCAL variable access, particularly in nested procedure *definitions*.

"FUNarg" problem of LISP:

```
(((lambda (x)
   (lambda(y)(+ x y)))
 3)
 4)
```

Python:

```
def f(x):
    return lambda y: x+y
z = f(3)(4)
```



Conventional solutions:

- Environments, closures.
- "static links" in stack frames, pointing to frames of statically enclosing blocks. This allows a run-time discipline which correctly accesses variables in enclosing blocks.

  BUT... enclosing block may no longer exist (as above!).

(C avoids this problem by outlawing nested procedure declarations!)

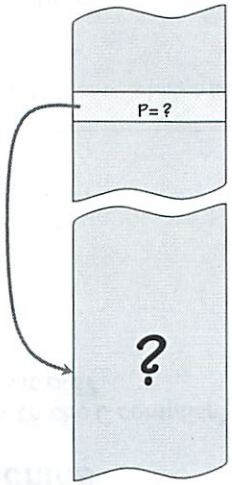
2. "Dangling References" - - -

Dangling References

global variable

```
int *p; /* a pointer */

int h(x)
{
   int y = x*3;
   p = &y;
   return 37;
}

h(10);
print(*p);
```

dereferenced



p can point to no where

What do we expect???

Randomness. Crashes. Smoke. Obscenities. Furious calls to Redmond, WA.

not ~~out~~ allowed in C

Dangling References:

different strokes...

C and C++: real tools, real dangers.

"You get what you deserve". but you can do it

Java / Python / ...: kiddie scissors only.

- No "ADDRESS OF" operator: language restrictions *forbid* constructs which could lead to dangling references.
- Automatic storage management: garbage collectors, reference counting: local variables allocated from a "heap" rather than a stack.

automatic type safety

"Safety" as a language/runtime property: guarantees against stray reads, writes.

- Tension: (manual) algorithm-specific optimization opportunites vs. simple, uniform, non-optimal storage management
- Tough language/compiler problem: abstractions, compiler technology that provides simple safety yet exploits efficiency of stack allocation.

Next Time: Building a Beta

figure out how this works

```
ack:    PUSH (LP)
        PUSH (BP)
        MOVE (SP, BP)
        PUSH (R1)
        PUSH (R2)
        LD (BP, -12, R2)
        LD (BP, -16, R0)
_L4:    SHLC (R0, 1, R1)
        BEQ (R2, _L1)
        ADDC (R2, 1, R1)
        BEQ (R0, _L1)
        SUBC (R2, 1, R1)
        SUBC (R0, 1, R0)
        PUSH (R0)
        PUSH (R2)
        BR (ack, LP)
        DEALLOCATE (2)
        MOVE (R1, R2)
        BR (_L4)
_L1:    MOVE (R1, R0)
        POP (R2)
        POP (R1)
        POP (BP)
        POP (LP)
        JMP (LP)
```

Recitation Handout

10/26

Problem 1

```
int gcd(int a,int b) {
  if (a == b) return a;
  if (a > b) return gcd(a-b,b);
  return gcd(a,b-a);
}
```

```
gcd:
        PUSH (LP)
        PUSH (BP)
        MOVE (SP, BP)                    ???
        PUSH (R1)                              0x00000594
        PUSH (R2)                              0x00001234
        LD (BP, -12, R0)                       0x00000046
        LD (BP, -16, R1)                       0x0000002A
        CMPEQ (R0, R1, R2)                     0x0000000E
        BT (R2, L1)                            0x0000001C
        CMPLE (R0, R1, R2)                     0x00000594
        BT (R2, L2)                            0x0000124C
        PUSH (R1)                         BP-->0x0000002A
        SUB (R0, R1, R2)                       0x0000000E
        PUSH (R2)                         SP-->0x00001254
        BR (gcd, LP)                           0x0000000E
        DEALLOCATE (2)
        BR (L1)
L2:
        SUB (R1, R0, R2)
        PUSH (R2)
        PUSH (R0)
        BR (gcd, LP)
        DEALLOCATE (2)
L1:
        POP (R2)
        POP (R1)
        MOVE (BP, SP)
        POP (BP)
        POP (LP)
        JMP (LP)
```

Problem 3

```
int gcd(int x, int y)
 {
   if (x == y) return x;
   if (y > x) y = y - x;
   else x = x - y;
   return gcd(x, y);
 }
```

```
gcd:     PUSH (LP)
         PUSH (BP)
         MOVE (SP, BP)
         PUSH (R1)
         PUSH (R2)
         LD (BP, -12, R1)
         LD (BP, -16, R2)
         CMPEQ (R2, R1, R0)
         BF (R0, ifxgty)
         MOVE (R1, R0)
         BR (done)
ifxgty:  CMPLE (R2, R1, R0)
         BT (R0, else)
         SUB (R2, R1, R2)
         BR (call)
else:    SUB (R1, R2, R1)
call:    PUSH (R2)
         PUSH (R1)
         BR (gcd, LP)
         DEALLOCATE (2)
done:    POP (R2)
         POP (R1)
         POP (BP)
         POP (LP)
         JMP (LP)
```

R0 = 0
R1 = 6
R2 = 3
...
BP = 0x00008058
LP = 0x80000068
SP = 0x00008070

Mem. Loc.	+0	+4	+8	+C
0x00008000		24	0x800000B0	
0x00008010				
0x00008020				
0x00008030				
0x00008040				
0x00008050				
0x00008060			0x80000068	0x8058
0x00008070	0xc0ffee	0xc0ffee	0xc0ffee	0xc0ffee

6.004 Recitation

10/26

Chris said Opcodes changed
└ redownload βsim for tutorial questions

Procedures

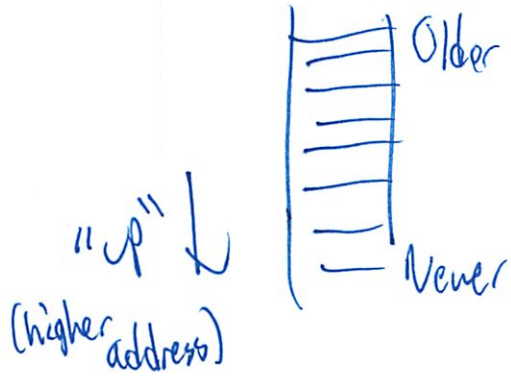
Associated w/ each procedural call are some things

BP → | arguments | ← # can be fixed or variable
| saved info | ← where to return
| local info |

- our convention: "callee saves"
- must restore registers to original value

most systems allocate a register for a base of frame pointer to stack frame (aka activation records)

Stack is best place to put it
└ is large global space in memory



②

Another register points to 1st unused place
"SP"

Can push or pop
↳ if you push, you must pop it back off

Fancy programs control memory allocated for stack

Problem 1

int gcd (int a, int b) {
↑ returns

3

Call:

BR(gcd, LP)
||
BEQ (R31, gcd, LP)

always 0, always happens

↑ where put the PC to return to (program counter +4)
"linkage pointer"

(3)

gcd:

Push(LP) ← remember addresses increment by 4

increments SP by 4
Stores LP into -4(SP)
↑ first you allocate
then you use it
↑ if you interrupt it
don't want bad state

Push(BP)
Move(SP, BP)
Push(R1)
Push(R2)

↑ we plan on using R1, R2 inside procedure

not planning other things

reverse order

exit:

Pop(R1)
Pop(R2)
Move(BP, SP)
POP(BP)
Pop(LP)
JMP(LP)

- This is boilerplate - need to copy + paste
  - GCC will optimize
  - can't jump somewhere for this - this is the jmp overhead!

(9)

Now I see why we store registers

- makes sense w/ stack and recursion or interior function calls
- if only 1 - perhaps less purpose
- but computers almost always have one thing calling another calling another, etc
- need to save their local variables
- want size to grow ∞ly
- So not $\underset{\text{fixed}}{\times}$ #s of registers)
- .

Stack is conceptually ∞

No fixed # of procedure calls

Now body of methods

if (x == y) return x

1st 2nd

x define y

CMPEQ(R2, R1, R0)

Need to load args first

our old arguments

LD(BP, -12, R1) | x

LD(BP, -16, R2) | y

BF(R0, if xgty)

Move(R1, R0)

BR(exit)

⋮

⑥ Calling it

return gcd(x, y)

Before

```
LD(BP, -16, R0)   ← load y
PUSH(R0)
LD(BP, -12, R0)
PUSH(R0)
```

call it

```
BEQ(R31, gcd, LP)
```

Get rid of arguments we put on earlier

```
DEALLOCATE(2)
    SUBC(SP, 8, SP)
```

↑ we are responsible for remaing the stuff we added

Why are we being so carefull deleting stuff?

- are doing some POPs before final restore
- must return it as you found it!
- or it will confusell others

Problem 1 more things

Problem 1 and 3 are what goes on quiz 3
└ for the last 30 years

Want binary for BR(L1)

- Same as BEQ(R31, L1, R31)

6	5	5	16
op	RC	RA	word offset literal

0110 1111 1111 0007
7 3 F F

↑ Count # instructions
Note Push is a macro - need to count as 2

Crib sheet will be provided

(8)

Stack question

GCD (27, 70)

Step 1 Label stack according to program

594	Old LP	1240
1234	Old BP	1250
46	R1	1254
2A	R2	1258
E	b	125C
1C	a	1260
594	Old LP	1264
1240	Old BP	
2A ← BP	R1	
E	R2	
1254 ← SP	Junk (where SP points and below)	

Previous caller ← Old BP

current BP

B value of current activation?

E

What is value of BP?

- where must stack be in memory???
- But we have Old BP
  - points to location containing 46 called 1240
- just count down!

6.004
Lab 5

10/27

Updated BSim to 1.2.0
Redoing Qu #3

9820F800 ✗

↑ Read wrong

77E1D002 ✓

What did I have before?

✓ Quiz

Checkoff

Usually people go

Push R3
2
1
⋮
Pop 1
2 → 3

Summary of β Instruction Formats

Operate Class:

Register	Symbol	Usage
R31	R31	Always zero
R30	XP	Exception pointer
R29	SP	Stack pointer
R28	LP	Linkage pointer
R27	BP	Base of frame pointer

31 – 26	25 – 21	20 – 16	15 – 11	10 – 0
10xxxx	Rc	Ra	Rb	*unused*

OP(Ra,Rb,Rc): Reg[Rc] ← Reg[Ra] op Reg[Rb]

Opcodes: **ADD** (plus), **SUB** (minus), **MUL** (multiply), **DIV** (divided by)
AND (bitwise and), **OR** (bitwise or), **XOR** (bitwise exclusive or), **XNOR** (bitwise exclusive nor),
CMPEQ (equal), **CMPLT** (less than), **CMPLE** (less than or equal) [result = 1 if true, 0 if false]
SHL (left shift), **SHR** (right shift w/o sign extension), **SRA** (right shift w/ sign extension)

31 – 26	25 – 21	20 – 16	15 – 0
11xxxx	Rc	Ra	literal (two's complement)

OPC(Ra,literal,Rc): Reg[Rc] ← Reg[Ra] op SEXT(literal)

Opcodes: **ADDC** (plus), **SUBC** (minus), **MULC** (multiply), **DIVC** (divided by)
ANDC (bitwise and), **ORC** (bitwise or), **XORC** (bitwise exclusive or), **XNORC** (bitwise exclusive nor)
CMPEQC (equal), **CMPLTC** (less than), **CMPLEC** (less than or equal) [result = 1 if true, 0 if false]
SHLC (left shift), **SHRC** (right shift w/o sign extension), **SRAC** (right shift w/ sign extension)

Other:

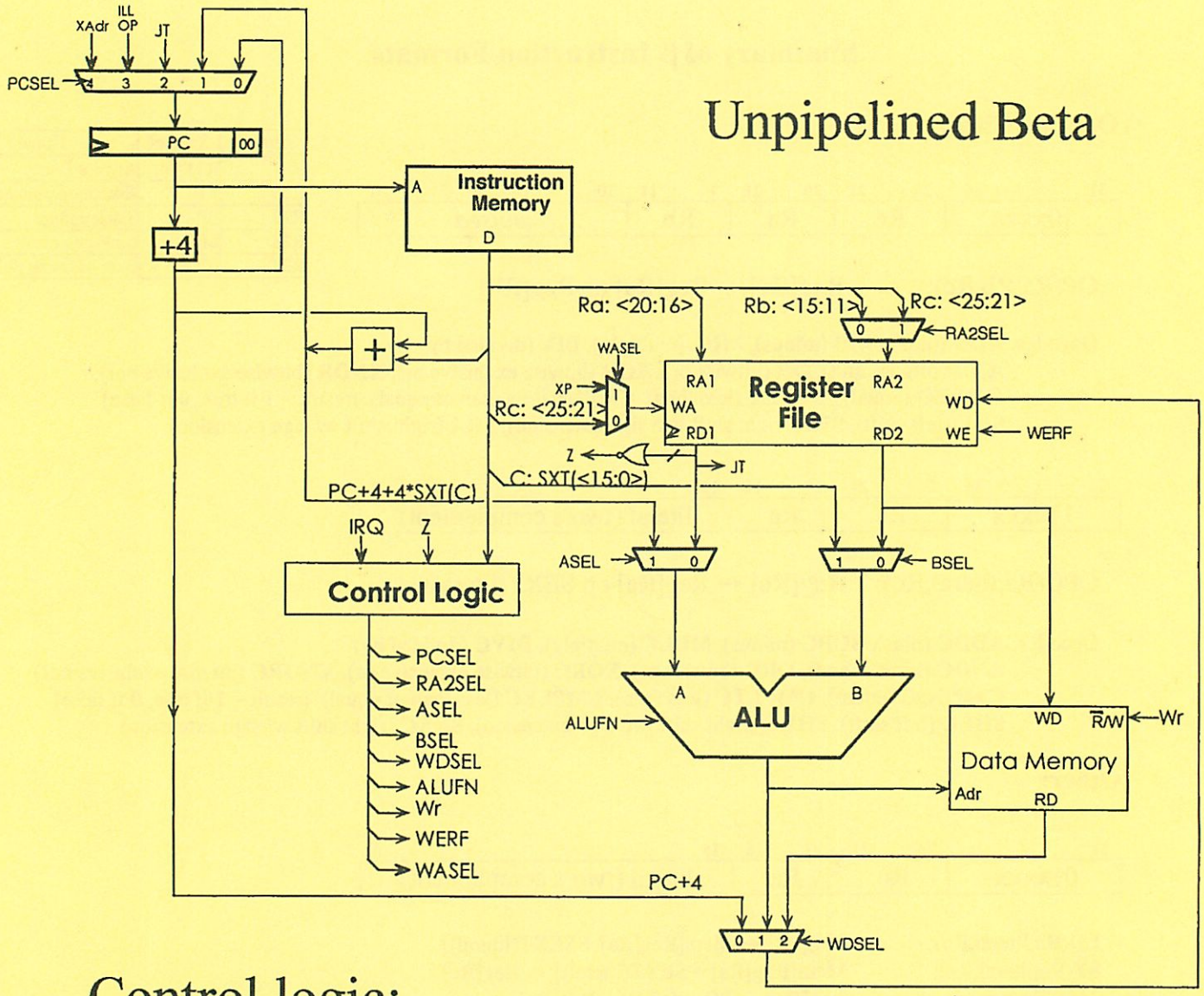
31 – 26	25 – 21	20 – 16	15 – 0
01xxxx	Rc	Ra	literal (two's complement)

LD(Ra,literal,Rc): Reg[Rc] ← Mem[Reg[Ra] + SEXT(literal)]
ST(Rc,literal,Ra): Mem[Reg[Ra] + SEXT(literal)] ← Reg[Rc]
JMP(Ra,Rc): Reg[Rc] ← PC + 4; PC ← Reg[Ra]
BEQ/BF(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] = 0 then PC ← PC + 4 + 4*SEXT(literal)
BNE/BT(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] ≠ 0 then PC ← PC + 4 + 4*SEXT(literal)
LDR(label,Rc): Reg[Rc] ← Mem[PC + 4 + 4*SEXT(literal)]

Opcode Table: (*optional opcodes)

2:0 / *5:3*	*000*	*001*	*010*	*011*	*100*	*101*	*110*	*111*
000								
001								
010								
011	**LD**	**ST**		**JMP**	**BEQ**	**BNE**		**LDR**
100	**ADD**	**SUB**	**MUL***	**DIV***	**CMPEQ**	**CMPLT**	**CMPLE**	
101	**AND**	**OR**	**XOR**	**XNOR**	**SHL**	**SHR**	**SRA**	
110	**ADDC**	**SUBC**	**MULC***	**DIVC***	**CMPEQC**	**CMPLTC**	**CMPLEC**	
111	**ANDC**	**ORC**	**XORC**	**XNORC**	**SHLC**	**SHRC**	**SRAC**	

Unpipelined Beta



Control logic:

	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	ILLOP	IRQ
ALUFN	F(op)	F(op)	A+B	A+B	--	--	--	A	--	--
WERF	1	1	1	0	1	1	1	1	1	1
BSEL	0	1	1	1	--	--	--	--	--	--
WDSEL	1	1	2	--	0	0	0	2	0	0
WR	0	0	0	1	0	0	0	0	0	0
RA2SEL	0	--	--	1	--	--	--	--	--	--
PCSEL	0	0	0	0	2	Z	~Z	0	3	4
ASEL	0	0	0	0	--	--	--	1	--	--
WASEL	0	0	0	--	0	0	0	0	1	1

6.004 Recitation

11/2

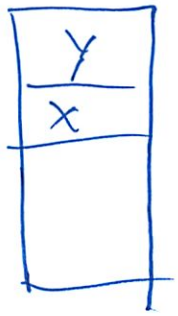
Quiz 3 Review

- all 3 quiz questions always similar
  - C program
  - stack trace
  - instruction set

#2 C program strategy

See 2 args

- put on stack
- should see frame code
  - push 2 args
  - BR(f, LP)
  - Deallocate(2)



- run through program
- then easy to do ~~[illegible]~~ questions

(2)

Some ??? marks in C program

↳ need to figure out

line up C line to ~~MAS~~ SY line

int a = (x+y) >> 1

LD X
LD Y
ADD
SRAC 1 ← (Can see it leaves in R2)

A

Can find values to registers

X → R1
A → R2
Y → R0

For loading arguments need BP −12

−16	Y
−12	X
−8	oldLP
−4	oldBP
BP →	

(3)

if (a == 0) return y ← Remember: leave in R0

BEQ (R2, done)

↑ goes to exit sequence

- deconstructs stack frame

(I didn't study, but worked a lot w/

- don't need to fully do it
- just a few qu
- and get ref material)

else return ???

└ must compute something into R0

lets look at Asy Code

SUB (R1, R2, R1)

└ R1 = R1 − R2

= x − a

PUSH(R1) ← x−a must be last argument

Push (R0)

BR(f, LP) → f(y, x−a)

Deallocate(2) ← remove arguments

(4)

(But still little things - like on jump store PC or PC+4)

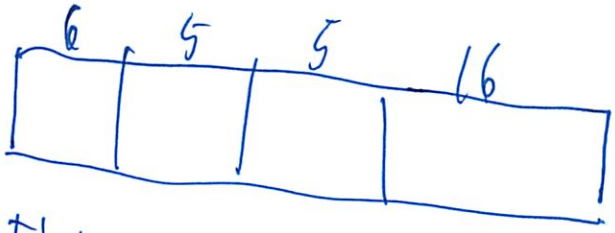
add (R2, R0, R0)

↳ so return is a + f(y, x-a)

(so recursion)

Now ans the qv

YY (BR, f, p)

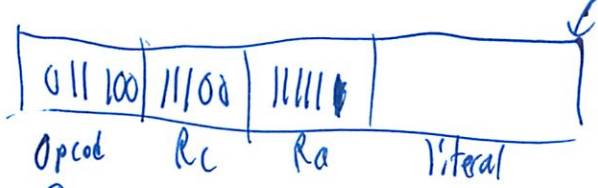


First expand macro

BEQ (R31, f, LP)

↳ R28

Ra Literal Rc



↑ read sheet 5:3 2:10

now need to count offset from here to f

-20 ← correct

-13

↳ don't multiply by 4

* count 2 for each push

(5)

So 20 00010100

Then complement 11101011
add 1 1

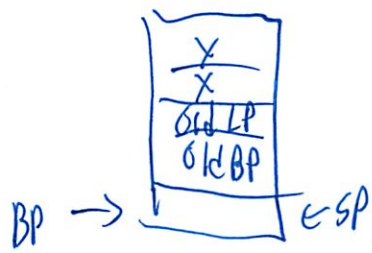
11101100) make negative

So final ans for literal

$\underbrace{1111}_{\text{pad w/ 1s}}\ \underbrace{1111}\ \underbrace{1110}\ \underbrace{1100}$

If we remove MOVE at ZZ:

So POP (R1) → R1
POP (R2) → R2



↑ same thing so no change

* Since all local variables have been removed

(6)

If you see a lable in midle of nowhere
↳ might be for quiz, not for Beta

Stack trace

1st thing you do: label purpose of each

y
x
old LP
old BP
BP → R1
R2

Recursive, so reuse lable

~~Return Ca~~ In normal nested call - similar but different

d) Argments for most recent

y=3, x=1

e) 0x234 is old BP
- look up 220

7

√ Terman said I labeled correctly

f) Original arguments x=7
y=6

g) What is val in LP

who was last person to put value into LP
look up the program
just coming back from recursive call
instruction following branch
└ the instruction of deallocate
how do we know?
look at other old LPs
54

h) What is value in R1
we see in old LP is 04
so that is original call
so BAD

(8)

#3 Can we just change the ROM

instructions currently are single lines implemented each time

~~alufn wef bsel wdsel wr Ra2sel Pcsel~~

	alufn	wef	bsel	wdsel	wr	Ra2sel	Pc sel	asel	wasel
Swap	No - impossible								
Str	"A"	–	–	–	1	1	0	1	–
bgt	No - impossible								

~~bge~~

Swap we can only write into 1 register

(ahh - I didn't know were to start but its actually fairly simple Stop + Think!)

Str Try writing what values to be

wef - not writing into register → don't care

wdsel → same → don't care

(9)

BGT Can't do

- subtract to tell if >
- but then can't test in the same instruction
  - can only test z at RD1
  - could add one at alu
  - is one - but not hooked up to control logic

All quizzes pretty similar

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Fall 2009

Quiz ~~#2~~ #3: November 6, 2009

Name		*Athena login name*	*Score*

TA: Caitlin	TA: Sandy	TA: Micah	TA: Sabrina
☐ WF 10, 26-322	☐ WF 11, 34-303	☐ WF 12, 34-304	☐ WF 1, 34-303
☐ WF 11, 26-322	☐ WF 12, 34-303	☐ WF 1, 34-304	☐ WF 2, 34-303

NOTE: There is reference material reproduced on the backs of quiz pages.

Problem 1 (5 points): **Quickies-but-Trickies**

(A) A Super Turing Machine is like a Turing Machine but has ***two*** infinite tapes rather than just one. Are there integer functions that are computable on an STM but not on a TM?

STM computes more functions? **Circle one: YES ... Can't tell ... NO**

(B) Turing machine TM_i halts when given *bounded* tape configuration k. Are there ***unbounded*** tape configurations for which TM_i will also halt?

Halts on unbounded tapes? **Circle one: YES ... Can't tell ... NO**

(C) A beta program contains the line

```
X:    LDR(X, R0)
```

After executing this instruction, is the low-order bit of the value loaded into **R0** a one?

Low order R0 bit is 1? **Circle one: YES ... Can't tell ... NO**

(D) In a Beta assembly-language program, the instruction

```
MULC(R0, 2*3*4, R1)
```

is replaced by

```
MULC(R0, 24, R1)
```

Does the modification make the resulting program smaller? Run faster?

Circle all that apply: Smaller ... Faster ... Neither

(E) In a standard Beta implementation, suppose you could speed up the generation of a single control signal in order to increase the Beta's clock frequency. Which signal would you choose to generate faster?

Timing-critical Beta control signal: _____

Problem 2. (13 points): **Software Reverse Engineering**

You are given the following listing of a C program and its translation to Beta assembly code:

```
int f(int x, int y) {
  int a = (x+y) >> 1;
  if (a == 0) return y;
  else return ???;
}
```

(Recall that *a* >> *b* means *a* shifted *b* bits to the right, propagating – ie, preserving -- sign)

```
f:   PUSH(LP)
     PUSH(BP)
     MOVE(SP, BP)
     PUSH(R1)
     PUSH(R2)
     LD(BP,-12,R1)
     LD(BP,-16,R0)
     ADD(R0, R1, R2)
     SRAC(R2, 1, R2)
xx:  BEQ(R2, bye)

     SUB(R1, R2, R1)
     PUSH(R1)
     PUSH(R0)
yy:  BR(f, LP)
     DEALLOCATE(2)
     ADD(R2, R0, R0)

bye: POP(R2)
     POP(R1)
zz:  MOVE(BP, SP)
     POP(BP)
     POP(LP)
     JMP(LP)
```

(A) (3 points) In the space below, fill in the binary value of the **BR** instruction stored at the location tagged '**yy:**' in the above program.

(fill in missing 1s and 0s for instruction at yy:)

(B) (1 point) Suppose the MOVE instruction at the location tagged '**zz:**' were eliminated from the above program. Would it continue to run correctly?

Still works fine? Circle one: YES ... NO

(C) (2 points) Give the missing expression designated by **???** in the C program above.

(Write missing C expression)

The procedure **f** is called from an external procedure and its execution is interrupted during a recursive call to **f**, just prior to the execution of the instruction tagged '**bye:**'. The contents of a region of memory are shown to the left.

NB: All addresses and data values are shown in hex. The **BP** register contains **0x250**, and **SP** contains **0x258**, and **R0** contains **0x5**.

	204:	CC	
	208:	4	
	20C:	7	
	210:	6	Y
	214:	7	X
	218:	E8	Old BP
	21C:	D4	Old LP
Old BP →	220:	BAD	R1
	224:	BABE	R2
	228:	1	Y
	22C:	6	X
	230:	54	Old LP
	234:	???	Old BP
Old BP →	238:	1	R1
	23C:	6	R2
	240:	3	Y
	244:	1	X
	248:	54	Old LP
	24C:	238	Old BP
BP->	250:	3	R1
	254:	3	R2
SP->	258:	-1	Junk

(D) (1 point) What are the arguments to the *most recent* active call to **f**?

Most recent arguments (HEX): x=0x_____; y=0x_____

(E) (1 point) What value is at stored at location 0x234, shown as **???** in the listing to the left?

Contents 0x234 (HEX): 0x_____

(F) (1 point) What are the arguments to the *original* call to **f**?

Original arguments (HEX): x=0x_____; y=0x_____

(G) (1 point) What value is in the **LP** register?

Contents of LP (HEX): 0x_____

(H) (1 point) What value was in **R1** at the time of the original call?

Contents of R1 (HEX): 0x_____

(I) (1 point) What value will be returned in **R0** as the value of the original call? [HINT: You can figure this out without getting the C code right!].

Value returned to original caller (HEX): 0x_____

(J) (1 point) What is the hex address of the instruction tagged "**yy:**"?

Address of yy (HEX): 0x_____

Problem 3 [7 points]: Demanding a Better Beta

Marketing has asked for the following instructions to be added to an Extended Beta instruction set, for implementation on an *unpipelined* Beta.

SWAPR(Rx, Ry) // Swap register contents
TMP ← Reg[Rx]
Reg[Rx] ← Reg[Ry]
Reg[Ry] ← TMP
PC ← PC + 4

STR(Rx, C) // Store relative
EA ← PC+4+4*SEXT(C)
Mem[EA] ← Reg[Rx]
PC ← PC + 4

BGT(Rx, Ry, C) // Branch if Greater
EA ← PC+4+4*SEXT(C)
If Reg[Rx] > Reg[Ry] then PC ← EA
else PC ← PC + 4

The Marketing people don't care about details of instruction coding (e.g., which fields are used to encode Rx and Ry in the above descriptions), but want to know which if any of the above can be implemented as a **single instruction** in the **existing Beta simply by changing the control ROM.**

Your job is to decide which of the above instructions can be implemented on the existing Beta, making appropriate choices for Rx and Ry, and to specify control signals that implement those instructions. You may wish to refer to the Beta diagram included among the reference material on backs of pages of this quiz.

For each instruction either fill in the appropriate values for the control signals in the table below or put a line through the whole row if the instruction cannot be implemented using the existing unpipelined Beta datapath. Use "--" to indicate a "don't care" value for a control signal.

Instr	ALUFN	WERF	BSEL	WDSEL	WR	RA2SEL	PCSEL	ASEL	WASEL
SWAPR									
STR									
BGT									

(Complete the above table)

END OF QUIZ!

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Fall 2009

Quiz #3: November 6, 2009

Name		*Athena login name*	*Score*
TA: Caitlin ☐ WF 10, 26-322 ☐ WF 11, 26-322	TA: Sandy ☐ WF 11, 34-303 ☐ WF 12, 34-303	TA: Micah ☐ WF 12, 34-304 ☐ WF 1, 34-304	TA: Sabrina ☐ WF 1, 34-303 ☐ WF 2, 34-303

NOTE: There is reference material reproduced on the backs of quiz pages.

Problem 1 (5 points): **Quickies-but-Trickies**

(A) A Super Turing Machine is like a Turing Machine but has *two* infinite tapes rather than just one. Are there integer functions that are computable on an STM but not on a TM?

STM computes more functions? **Circle one: YES ... Can't tell ... NO**

(B) Turing machine TM_i halts when given *bounded* tape configuration k. Are there *unbounded* tape configurations for which TM_i will also halt?

Halts on unbounded tapes? **Circle one: YES ... Can't tell ... NO**

(C) A beta program contains the line

```
X:    LDR(X, R0)
```

After executing this instruction, is the low-order bit of the value loaded into **R0** a one?

Low order R0 bit is 1? **Circle one: YES ... Can't tell ... NO**

(D) In a Beta assembly-language program, the instruction

```
MULC(R0, 2*3*4, R1)
```

is replaced by

```
MULC(R0, 24, R1)
```

Does the modification make the resulting program smaller? Run faster?

Circle all that apply: Smaller ... Faster ... Neither

(E) In a standard Beta implementation, suppose you could speed up the generation of a single control signal in order to increase the Beta's clock frequency. Which signal would you choose to generate faster?

Timing-critical Beta control signal: RA2SEL

Problem 2. (13 points): **Software Reverse Engineering**

You are given the following listing of a C program and its translation to Beta assembly code:

```
int f(int x, int y) {
  int a = (x+y) >> 1;
  if (a == 0) return y;
  else return ???;
}
```

(Recall that $a >> b$ means a shifted b bits to the right, propagating – ie, preserving -- sign)

```
f:   PUSH(LP)
     PUSH(BP)
     MOVE(SP, BP)
     PUSH(R1)
     PUSH(R2)
     LD(BP,-12,R1)
     LD(BP,-16,R0)
     ADD(R0, R1, R2)
     SRAC(R2, 1, R2)
xx:  BEQ(R2, bye)

     SUB(R1, R2, R1)
     PUSH(R1)
     PUSH(R0)
yy:  BR(f, LP)
     DEALLOCATE(2)
     ADD(R2, R0, R0)

bye: POP(R2)
     POP(R1)
zz:  MOVE(BP, SP)
     POP(BP)
     POP(LP)
     JMP(LP)
```

(A) (3 points) In the space below, fill in the binary value of the **BR** instruction stored at the location tagged '**yy:**' in the above program.

0 1 1 1 0 1	1 1 1 0 0	1 1 1 1 1	1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 0

(fill in missing 1s and 0s for instruction at yy:)

(B) (1 point) Suppose the MOVE instruction at the location tagged '**zz:**' were eliminated from the above program. Would it continue to run correctly?

Still works fine? Circle one: YES ... NO

(C) (2 points) Give the missing expression designated by **???** in the C program above.

a + f(y, x-a)

(Write missing C expression)

The procedure **f** is called from an external procedure and its execution is interrupted during a recursive call to **f**, just prior to the execution of the instruction tagged '**bye:**'. The contents of a region of memory are shown to the left.

NB: All addresses and data values are shown in hex. The **BP** register contains **0x250**, and **SP** contains **0x258**, and **R0** contains **0x5**.

```
      204:   CC
      208:    4
      20C:    7
      210:    6
      214:    7
      218:   E8
      21C:   D4
      220:  BAD
      224: BABE
      228:    1
      22C:    6
      230:   54
      234:  ???
      238:    1
      23C:    6
      240:    3
      244:    1
      248:   54
      24C:  238
BP->  250:    3
      254:    3
SP->  258:   -1
```

(D) (1 point) What are the arguments to the *most recent* active call to **f**?

Most recent arguments (HEX): x=0x 1 **; y=0x** 3

(E) (1 point) What value is at stored at location 0x234, shown as **???** in the listing to the left?

Contents 0x234 (HEX): 0x 220

(F) (1 point) What are the arguments to the *original* call to **f**?

Original arguments (HEX): x=0x 7 **; y=0x** 6

(G) (1 point) What value is in the **LP** register?

Contents of LP (HEX): 0x 54

(H) (1 point) What value was in **R1** at the time of the original call?

Contents of R1 (HEX): 0x BAD

(I) (1 point) What value will be returned in **R0** as the value of the original call? [HINT: You can figure this out without getting the C code right!].

Value returned to original caller (HEX): 0x E (= 5 + 3 + 6)

(J) (1 point) What is the hex address of the instruction tagged "yy:"?

Address of yy (HEX): 0x 50

Problem 3 [7 points]: Demanding a Better Beta

Marketing has asked for the following instructions to be added to an Extended Beta instruction set, for implementation on an *unpipelined* Beta.

SWAPR(Rx, Ry) // Swap register contents
TMP ← Reg[Rx]
Reg[Rx] ← Reg[Ry]
Reg[Ry] ← TMP
PC ← PC + 4

STR(Rx, C) // Store relative
EA ← PC+4+4*SEXT(C)
Mem[EA] ← Reg[Rx]
PC ← PC + 4

BGT(Rx, Ry, C) // Branch if Greater
EA ← PC+4+4*SEXT(C)
If Reg[Rx] > Reg[Ry] then PC ← EA
else PC ← PC + 4

The Marketing people don't care about details of instruction coding (e.g., which fields are used to encode Rx and Ry in the above descriptions), but want to know which if any of the above can be implemented as a **single instruction** in the **existing Beta simply by changing the control ROM**.

Your job is to decide which of the above instructions can be implemented on the existing Beta, making appropriate choices for Rx and Ry, and to specify control signals that implement those instructions. You may wish to refer to the Beta diagram included among the reference material on backs of pages of this quiz.

For each instruction either fill in the appropriate values for the control signals in the table below or put a line through the whole row if the instruction cannot be implemented using the existing unpipelined Beta datapath. Use "--" to indicate a "don't care" value for a control signal.

Instr	ALUFN	WERF	BSEL	WDSEL	WR	RA2SEL	PCSEL	ASEL	WASEL
~~SWAPR~~									
STR	A	0	--	--	1	1	0	1	--
~~BGT~~									

(Complete the above table)

END OF QUIZ!

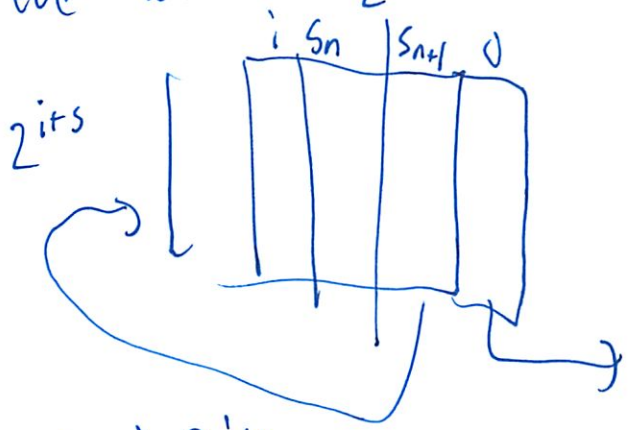
6.004 Topic Review

Programability

FSMs

i input
o output
s state

Need ROM 2^{i+s} words · $(o+s)$ bits



$2^{(o+s)\,2^{i+s}}$ possible FSMs

Can describe program as FSM
Must be finite # of steps

Turing Machine

Solves ∞ problem of FSM
Doably ∞ tape
Finite alphapet
Out — write
— move

② Has starting state, halt state

Give a name to tape and machine

$$y = T_i[x]$$

If computable - can build w/ Turing machine

$$f(x) = T_k[x] = f_k(x)$$

Some things not computable

- like when a fn will halt

Universal Turing machine

$$U(k, j) = T_k[j]$$

machine tape

k encodes a program

j encodes input data

T_U interprets program

$$\begin{matrix} k \rightarrow \\ j \rightarrow \end{matrix} \boxed{T_U} \xrightarrow{T_k[j]}$$

Compilers better than building new hw each time

(3)

Designing an Instruction Set (skipped class)

Design data path to compute value

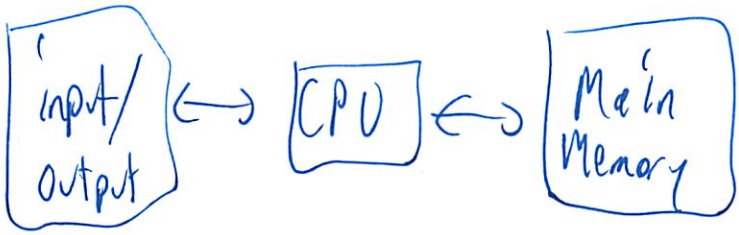
Rewrite as steps - break up in parts

Write values needed at each part

* Programability lets you reuse parts

Want
1. A good amt of storage
2. A good # of operations
3. Ability to generate new programs + execute

Von Neumann



We have β architecture

Engineering Choices

Tradeoffs

Uniformity

Complexity

(4)

(Not writing parts of Beta - are on cheat sheet)

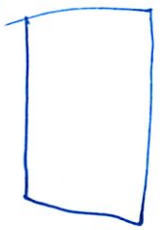
LD (ra, const, rc)

Mem[Reg[Ra] + sxt (c)] → Reg[Rc]

St (rc, cont, ra)

Reg[Rc] → Mem[Reg(Ra) + sxt (c)]

Memory



Register
- temp

<u>Access methods</u>

Absolute
- constant

Indirect
- Reg[Ra]

Displacement
- Reg[Ra] + sxt(c)

Need loops
└ So need to branch

(5)

BEQ(ra, label, rc)
- Branch if ra = 0

offset = lable − (address BEQ) / 4 − 1 *remember

If trying to figure out FSMs − try it!

From reading recitation − anything about halting doesn't work

Load enabled registers

Moore SM − output only depends on current state
Mealy SM − output depends on current state + input

ML, Assemblers, Compilers

Interpeters − single program which mimics behavior of easier machine
− sometimes several layers of
− at run time

Compilers translate to harder to program − but closer to machine

(6)

UASM - 6.004 assembler

a = next address to be filled

"X:" is "X = ."

Macro instructions
- Short hand
- some are >1 - need to see which

BR() always branch

BR(Label, RC) store branch from +4 in RC

Call(Lable) = BEQ(R31, Label, LP)

Push, POP are 2 instructions

C is higher level lang
↳ combiled

Arrays stored
- calc offset

~~Optimize~~
~~Loops~~ ~~Lwhile~~ ~~Clex~~

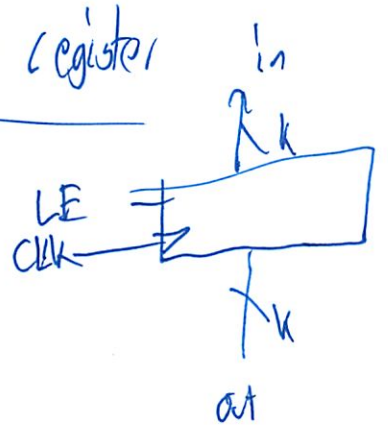
7

<u>Loops</u>

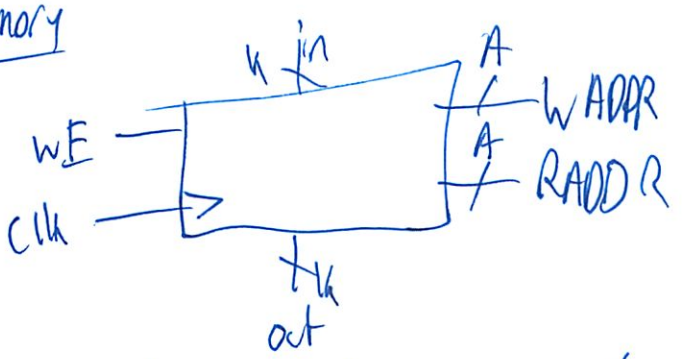
```
Lwhile
    (expr)
    BF (rx, Lend while)
    comple
    BR (L while)

Lend while:
```

Optimize: move store, loads out of loops

<u>k-bit register</u>



<u>Memory</u>



size $2^A \cdot k$

↑ k-bit registers stacked on top of

(8)

R31 is always 0

Long() shortcut ← (have not studied - hope they don't ask)

Pre-con states

- not really in here

Long() splits into byte sized chunks

Registers

R31	0	
R30	XP	exception
R29	SP	stack
R28	LP	link
R27	BP	base of frame

Allocate + Deallocate are 1 instruction each

<u>Standard prologue</u>

Push (LP)
Push (BP)) from old program
Move (SP, BP)
Allocate (lx) ← space for locals (I am unsure about this one)
Push regs to store

(9)

Standard epilogue

(pop registers used in procedure)

Move (BP, SP) ← deallocates space for locals

POP (BP)

POP (LP)

JMP (LP)

Bitwise NOT is $-x-1$

Shift

ALS

discard ← ← ← ←

← 0

multiply by 2^n

ARS divide by 2^n and rounding to $-\infty$

0

↓ ↓ ↓ ↓ → discard

0 0

↑ ↑

copy MSB

Logical

Left - same

Right - insert 0 instead of copying MSB

↑ 1st bit

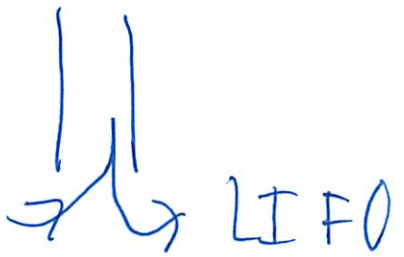
(10)

>> << are logical shifts

Stacks + Procedures

- reusable code fragments
- packaged up

put data on stack



Push args on in reverse order

To access kth local variable k > 4

jth argument -4 : (j+3)

Result left in R0

Recursion - crawl up stack to search

(11)

Building the Beta

trade off performance / price

$$MIPS = \frac{\text{MHz Clock Freq}}{\text{Clocks per instructions}}$$

Incremental featerism to build

Multiport Reg ~~file~~ file
- read 2 at once
- 1 write

Reading maintains value till next clock edge

PC to determine step point

Gets instruction from memory

JMP instructions

Exceptions
- ~~load~~ load exception vector
- Save PC +4 into R30 (xp)

(supervisor bit not covered much)

Review Lab 5 Qu

11/4

Remember OP(ra, rb, rc)

op → save

OR is bitwise

shift

But where are instructions stored in memory

Starts 000 here - but is that always true?
w/ 1st

Look at bulls + cows

which has checkoff loaded in

0000 BR (I_Reset)

0004 BR (I_Illop

0008 BR (I_Clk)

0012 BR (I_kbd)

0010 Halt()

Then others defined below

so how do we know on quiz?

2

Remember assempler steps - converts lables to memory locations

(Think I pretty much got it - it was just unsuity w/ wrong program)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Spring 2010

Quiz #3: April 9, 2010

Name		*Athena login name*	*Score*
TA: Kevin	TA: Erica	TA: Sabrina	TA: Nicole
☐ WF 10, 34-303	☐ WF 12, 34-302	☐ WF 1, 34-301	☐ WF 2, 34-302
☐ WF 11, 34-303	☐ WF 1, 34-302	☐ WF 2, 34-301	☐ WF 3, 34-302

NOTE: there is reference material on the reverse sides of quiz pages.

Problem 1 (5 points): **Quickies-but-Trickies**

(A) We saw that a universal Turing Machine can emulate the behavior of any other Turing machine. Is there a universal FSM that can emulate any other FSM?

Does there exist a universal FSM? **Circle one: YES ... Can't tell ... NO**

(B) Let **F(Y)** be 1 if Obama is president during any portion of the year **Y**, and 0 otherwise. Is **F** a computable function?

F computable? **Circle one: YES ... Can't tell ... NO**

don't really have formal understanding why!
why not

(C) A beta program contains the line

```
X:      BR(X)
```

What value is in the lower 16 bits (literal field) of the **BR** instruction? Answer in hex.

Low 16 bits of BR instruction at X: 0x ~~0012~~ FFFF

? start at 12

why??

(D) (1 point) A Beta assembly-language program contains, as the first executable instruction, a line containing **LONG(0xFFFFFFFF)**. Will this cause a translation-time error? A run-time error?

Circle one: Translation Error ... Runtime error ... No Error

why?

(E) In a standard Beta implementation, suppose you could speed up the generation of a single control signal in order to increase the Beta's clock frequency. Which signal would you choose to generate faster?

? No good real one

Timing-critical Beta control signal: ~~PCSel~~ RA2SEL

← no PCSel

why?

Chris' Strategy

put args on stack

BR(f, LP)

deallocate(2)

Problem 2. (12 points): **Software Detective**

You are given the following listing of a C program and its translation to Beta assembly code:

```
int f(int x, int y) {
  int a = x-1, b = x+y;
  if (x == 0) return y;
  return f(a, ???);
}
```

run through program

x = R1 = new Y
a = R2 = new X

f(a, X)

Y
X
OldLP
OldBP
Saved R1
Saved R2

← BP

← SP

```
f:      PUSH(LP)
        PUSH(BP)
mm:     MOVE(SP, BP)
        PUSH(R1)
        PUSH(R2)          save
        LD(BP, -16, R0)   Y
yy:     LD(BP, -12, R1)   X
        BEQ(R1, xx)       if X=0
        SUBC(R1, 1, R2)   a
        ADD(R0, R1, R1)   X+Y=X
        PUSH(R1)          save X,a
        PUSH(R2)
        BR(f, LP)
zz:     DEALLOCATE(2)
        LD(BP, -16, R1)
        ADD(R1, R0, R0)
        PUSH(R0)
ww:     PUSH(R2)
        BR(f, LP)
        DEALLOCATE(2)
xx:     POP(R2)
        POP(R1)
        POP(BP)
        POP(LP)
        JMP(LP)
```

why → return

12
1100
11111 0011

(A) (2 points) In the space below, fill in the binary value of the **LD** instruction stored at the location tagged '**yy:**' in the above program.

LD Rc 27 −12

011000	00001	11011	1111111111110100

how do −12 again − complement add 1

(fill in missing 1s and 0s for instruction at yy:) reverse

(B) (1 point) Suppose the MOVE instruction at the location tagged '**mm:**' were eliminated from the above program. Would it continue to run correctly?

move is a macro. I did not write down

was yes before − why

Still works fine? Circle one: YES ... Can't tell ... NO

worked before since BP=SP − all local variables remaint (why not here?)

(C) (2 points) Give the missing expression designated by ??? in the C program above.

return f(a, X) **(Write missing C expression)**

do they want a letter or entire expression

Y + f(a,b)

− seems like just letter

To need full out expression

− should have done through whole thing

Stack Detective

The procedure **f** is called from location 0xFC and its execution is interrupted during a recursive call to **f**, just prior to the execution of the instruction tagged '**xx:**'. The contents of a region of memory, including the stack, are shown to the left.

NB: All addresses and data values are shown in hex. The **BP** register contains **0x494,** and **SP** contains **0x49C.**

448:	2	
44C:	4	
450:	7	
454:	3	y
458:	2	x
45C:	100	old LP
460:	D4	old BP
BP → 464:	3	saved R2
468:	4	saved R1
46C:	5	y
470:	1	x
474:	50	old LP
478:	???	old BP
47C:	5	saved R2
480:	1	saved R1
484:	B	y
488:	0	x
48C:	70	old LP
490:	47C	old BP
BP->494:	5	saved R2
498:	0	saved R1
SP->49C:		

(D) (1 point) What are the arguments to the *most recent* active call to **f**?

Most recent arguments (HEX): x=0x 0 **; y=0x** B

(E) (1 point) What value is stored at location **0x478**, shown as **???** in the listing to the left?

1. Contents 0x478 (HEX): 0x 464

(F) (1 point) What are the arguments to the *original* call to **f**?

Original arguments (HEX): x=0x 2 **; y=0x** 3

(G) (1 point) What value is in the **LP** register?

Contents of LP (HEX): 0x 70

(H) (1 point) What value was in **R1** at the time of the original call?

Contents of R1 (HEX): 0x ~~4~~ 3 ?? should it be 2?

(I) (1 point) What value is in **R0**?

Value currently in R0 (HEX): 0x B

(J) (1 point) What is the hex address of the instruction tagged "**ww:**"?

Address of ww (HEX): 0x ~~28~~ 64

We're not at start
Can calc relative offset
On 2 branches
Count up from 50
*Remember label points to line
Branch points to next line

Problem 3 (8 Points): **Beta control signals**

Following is an incomplete table listing control signals for several instructions on an unpipelined Beta. You may wish to consult Beta diagram and instruction set summary attached for your reference on the reverse side of exam pages.

The operations listed include two existing instructions and three proposed additions to the Beta instruction set:

LDX(Ra, Rb, Rc) // **Load, double indexed**
EA ← Reg[Ra] + Reg[Rb]
Reg[Rc] ← Mem[EA]
PC ← PC + 4

MVZC(Ra, literal, Rc) // **Move constant if zero**
If Reg[Ra] == 0 then Reg[Rc] ← SXT(literal)
PC ← PC + 4

STR(Rc, C) // **Store relative**
EA ← PC+4+4*SEXT(C)
Mem[EA] ← Reg[Rc]
PC ← PC + 4

In the following table, **--** represents a "**don't care**" or unspecified value; **Z** is the value (0 or 1) output by the 32-input NOR in the unpipelined Beta diagram. Your job is to complete the table, by filling in each unshaded entry. In each case, enter an opcode, a value, an expression, or **--** as appropriate.

See what values it would need to be

Chart is a pain - sk'll

Instr	ALUFN	WERF	BSEL	WDSEL	WR	RA2SEL	PCSEL	ASEL	WASEL
JMP	--	1	--	0	0	--	2	--	0
BEQ	--	1	--	0	0	--	Z	--	0
LDX	A+B	1	0	2	0	0	0	0	0
MVZC	B	Z	1	1	0	--	0	--	0
STR	A	0	--	--	1	1	0	1	--

(Complete the above table)

So here all were possible

End of Quiz!

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Spring 2010

Quiz #3: April 9, 2010

Name Solutions		*Athena login name*	*Score*
TA: Kevin ☐ WF 10, 34-303 ☐ WF 11, 34-303	TA: Erica ☐ WF 12, 34-302 ☐ WF 1, 34-302	TA: Sabrina ☐ WF 1, 34-301 ☐ WF 2, 34-301	TA: Nicole ☐ WF 2, 34-302 ☐ WF 3, 34-302

NOTE: there is reference material on the reverse sides of quiz pages.

Problem 1 (5 points): **Quickies-but-Trickies**

(A) We saw that a universal Turing Machine can emulate the behavior of any other Turing machine. Is there a universal FSM that can emulate any other FSM?

Does there exist a universal FSM? **Circle one: YES ... Can't tell .. NO**

(B) Let **F(Y)** be 1 if Obama is president during any portion of the year **Y**, and 0 otherwise. Is **F** a computable function?

F computable? **Circle one: YES .. Can't tell ... NO**

(C) A beta program contains the line

```
X:    BR(X)
```

What value is in the lower 16 bits (literal field) of the **BR** instruction? Answer in hex.

Low 16 bits of BR instruction at X: 0x FFFF

(D) (1 point) A Beta assembly-language program contains, as the first executable instruction, a line containing **LONG(0xFFFFFFFF)**. Will this cause a translation-time error? A run-time error?

Circle one: Translation Error ... Runtime error ... No Error

(E) In a standard Beta implementation, suppose you could speed up the generation of a single control signal in order to increase the Beta's clock frequency. Which signal would you choose to generate faster?

Timing-critical Beta control signal: RA2SEL

Problem 2. (12 points): **Software Detective**

You are given the following listing of a C program and its translation to Beta assembly code:

```
int f(int x, int y) {
 int a = x-1, b = x+y;
 if (x == 0) return y;
 return f(a, ???);
}
```

```
f:      PUSH(LP)
        PUSH(BP)
mm:     MOVE(SP, BP)
        PUSH(R1)
        PUSH(R2)
        LD(BP, -16, R0)
yy:     LD(BP, -12, R1)
        BEQ(R1, xx)
        SUBC(R1, 1, R2)
        ADD(R0, R1, R1)
        PUSH(R1)
        PUSH(R2)
        BR(f, LP)
zz:     DEALLOCATE(2)
        LD(BP, -16, R1)
        ADD(R1, R0, R0)
        PUSH(R0)
ww: 64  PUSH(R2)
    6C  BR(f, LP)
    70  DEALLOCATE(2)
xx:     POP(R2)
        POP(R1)
        POP(BP)
        POP(LP)
        JMP(LP)
```

(A) (2 points) In the space below, fill in the binary value of the **LD** instruction stored at the location tagged '**yy:**' in the above program.

0 1 1 0 0 0	0 0 0 0 1	1 1 0 1 1	1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0

(fill in missing 1s and 0s for instruction at yy:)

(B) (1 point) Suppose the MOVE instruction at the location tagged '**mm:**' were eliminated from the above program. Would it continue to run correctly?

Still works fine? Circle one: YES ... Can't tell .. NO

(C) (2 points) Give the missing expression designated by **???** in the C program above.

Y + f(a, b)

(Write missing C expression)

The procedure **f** is called from location 0xFC and its execution is interrupted during a recursive call to **f**, just prior to the execution of the instruction tagged '**xx:**'. The contents of a region of memory, including the stack, are shown to the left.

NB: All addresses and data values are shown in hex. The **BP** register contains **0x494**, and **SP** contains **0x49C**.

```
448:    2
44C:    4
450:    7
454:    3    y
458:    2    x
45C:  100   LP
460:   D4   BP
464:    3   R1
468:    4   R2
46C:    5    y
470:    1    x
474:   50   LP
478:  ???   BP
47C:    5   R1
480:    1   R2
484:    B    y
488:    0    x
48C:   70   LP
490:  47C   BP
BP->494:  5  R1
498:    0   R2
SP->49C:
```

(D) (1 point) What are the arguments to the *most recent* active call to **f**?

Most recent arguments (HEX): x=0x 0 **; y=0x** B

(E) (1 point) What value is stored at location **0x478**, shown as **???** in the listing to the left?

1. Contents 0x478 (HEX): 0x 464

(F) (1 point) What are the arguments to the *original* call to **f**?

Original arguments (HEX): x=0x 2 **; y=0x** 3

(G) (1 point) What value is in the **LP** register?

Contents of LP (HEX): 0x 70

(H) (1 point) What value was in **R1** at the time of the original call?

Contents of R1 (HEX): 0x 3

(I) (1 point) What value is in **R0**?

Value currently in R0 (HEX): 0x B

(J) (1 point) What is the hex address of the instruction tagged "**ww:**"?

Address of ww (HEX): 0x 64

Problem 3 (8 Points): **Beta control signals**

Following is an incomplete table listing control signals for several instructions on an unpipelined Beta. You may wish to consult Beta diagram and instruction set summary attached for your reference on the reverse side of exam pages.

The operations listed include two existing instructions and three proposed additions to the Beta instruction set:

LDX(Ra, Rb, Rc) **// Load, double indexed**
EA ← Reg[Ra] + Reg[Rb]
Reg[Rc] ← Mem[EA]
PC ← PC + 4

MVZC(Ra, literal, Rc) **// Move constant if zero**
If Reg[Ra] == 0 then Reg[Rc] ← SXT(literal)
PC ← PC + 4

STR(Rc, C) **// Store relative**
EA ← PC+4+4*SEXT(C)
Mem[EA] ← Reg[Rc]
PC ← PC + 4

In the following table, -- represents a "**don't care**" or unspecified value; **Z** is the value (0 or 1) output by the 32-input NOR in the unpipelined Beta diagram. Your job is to complete the table, by filling in each unshaded entry. In each case, enter an opcode, a value, an expression, or -- as appropriate.

Instr	ALUFN	WERF	BSEL	WDSEL	WR	RA2SEL	PCSEL	ASEL	WASEL
JMP	--	1	--	0	0	--	2	--	0
BEQ	--	1	--	0	0	--	Z	--	0
LDX	A+B	1	0	2	0	0	0	0	0
MVZC	B	Z	1	1	0	--	0	--	0
STR	A	0	--	--	1	1	0	1	--

(Complete the above table)

End of Quiz!

1	/ 4
2	/ 11
3	/ 8
4	/ 7
	/ 30

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Fall 2010

Quiz #3: November 5, 2010

Name		*Athena login name*	*Score*
TA: Caitlin, 26-322 ☐ WF 10 ☐ WF 11	TA: Quentin, 34-303 ☐ WF 11 ☐ WF 12	TA: Sabrina, 34-304 ☐ WF 12 ☐ WF 1	TA: Steve, 34-303 ☐ WF 2

Note: There is reference material, of possible use in solving problems on this quiz, on the backs of quiz pages.

Problem 1 (4 points): **Wish I'd stayed awake during that lecture ...**

For each of the following statements, indicate whether the statement is

TRUE: proven fact; or
ACCEPTED: unproven but generally accepted as true; or
UNKNOWN: neither proven nor disproven; or
FALSE: known to be untrue.

(A) A Universal Turing Machine can compute every well-defined integer function.

Circle one: ... TRUE ... ACCEPTED ... UNKNOWN ... FALSE

Shannon

(B) Every well-defined integer function that can be computed by any digital computer that could possibly be built is computable by some Turing machine.

Circle one: ... TRUE ... ACCEPTED ... UNKNOWN ... FALSE

Oh never prove'n

(C) Bertrand Russell is the Pope.

Circle one: ... TRUE ... ACCEPTED ... UNKNOWN ... FALSE

(D) In mathematics, there are statements that are both well-defined and true, but which cannot be proven.

true means prove

Circle one: ... TRUE ... ACCEPTED ... UNKNOWN ... FALSE

Contradiction

I disagree here...

Problem 2 (11 points): **Reverse Engineering a Mystery Function**

You've been commissioned by a government agency to reverse-engineer a mysterious procedure found on the disk of a Beta system used by a cyber-terrorist cell. You've given an incomplete copy of the C source language for the function **f** (shown below), as well as its complete translation to Beta assembly code shown to the right:

```
// Mystery function:

int f(int x) {
  int a = x & 5;   // bitwise AND

  if (x == 0) return 0;
  else return ?????;
}
```

```
f:    PUSH(LP)
      PUSH(BP)
      MOVE(SP, BP)
      ALLOCATE(1)   ← leave space for 1 local
      PUSH(R1)

      LD(BP, -12, R0)   x variable
      ANDC(R0, 5, R1)   a=
      ST(R1, 0, BP)     store R1(a) in BP

xx:   BEQ(R0, bye)

      SUBC(R0, 1, R0)   X = X-1
      PUSH(R0)          put on stack
yy:   BR(f, LP)         call function
      DEALLOCATE(1)

      LD(BP, 0, R1)     LD BP into R1
      ADD(R1, R0, R0)   BP + x = R0 x
bye:  POP(R1)           ← remove
      MOVE(BP,SP)
      POP(BP)
      POP(LP)           R0 = X returned
      JMP(LP)
```

WTF? → ADD(R1, R0, R0)

leave

Note: you may wish to refer to reference material on the backs of quiz pages.

(A) (2 points) Give the missing expression shown in the C code as "**?????**"

return at f(x-1)

Write C expression for ????? above

(B) (1 point) Is the value of the local variable **a** stored in the stack frame of the Beta program? If so, give its offset relative to the contents of **BP**; otherwise, write "None":

Oh R1 = a

Offset of a, or None: 0

if it did not get

(C) (3 points) Give the 32-bit binary translation of the **BR** instruction tagged **yy:**:

Need macro shortcut

BEQ(R1, f, LP)

(fill in 1s and 0s above)

I need f pointer (relative from before)

Problem 2 (continued)

```
    13C:     7
    140:     7
    144:    5C
    148:    D4
    14C:     5
    150:     3
    154:     6
    158:    A4
    15C;   14C
    160:     4
    164:     5
    168:     5
    16C:    A4
    170:   160
BP->174:     5
    178:     4
```

The function **f** is called from an external main program, and the machine is halted when a recursive call to **f** is about to execute the **BEQ** instruction tagged **xx:.** The **BP** register of the halted machine contains **0x174,** and the hex contents of a region of memory location are shown to the left.

(D) (1 point) What is the value in **SP**?

HEX contents of SP: 0x_____

(E) (1 point) What is the value stored in the local variable **a** in the current stack frame?

HEX value of a: 0x_____

(F) (1 point) What is the address of the **BR** instruction that made the original call to **f** from the external main program?

Address of BR for original call: 0x_____

(G) (1 point) What value is currently in the **PC**?

HEX contents of PC: 0x_____

The summer intern working for you, after looking at the assembly code for **f**, argues that the cyber-terrorist group that wrote this code isn't very clever. He argues that one could simply delete four instructions from the assembly-language program -- an **LD**, an **ST**, an **ALLOCATE**, and a **MOVE** -- and the program would continue to work as before (but faster and using less space).

(H) (1 point) Is he right? Can one in fact delete four lines of code as described and still have a working program?

Can one optimize by deleting 4 such lines? Circle one: YES ... NO

Problem 3 (8 points): **A Better Beta**

Marketing has decided that the next model Beta needs several additional instructions, and has called you in as a consultant to decide, in each case, whether

(i) the instruction can be implemented simply as a macro, whose body contains a **single** existing Beta instruction that performs the indicated operation;

(ii) the instruction can be implemented using the existing data paths, a new opcode and appropriate control signal generation to the Beta's control ROM; or

(iii) the instruction cannot be implemented without changes to the Beta's data paths.

For each of the following proposed new instructions, you are to determine whether it can be translated (using a macro) to a single existing instruction, and, if so, to write the equivalent assembly language instruction. If it can't be translated to an existing instruction, you must determine whether it can be implemented as a new opcode using existing Beta data paths (including your ALU from Lab 3), and, if so, to specify appropriate control signals for that opcode. If neither implementation strategy will implement the indicated operation, circle NONE and give a brief (several-word) explanation.

The Beta implementation you are given is precisely that shown in lecture; its diagram, control signals, and instruction set are given for reference on the backs of pages of this quiz. Note that the ALU is the one you implemented in Lab 3, although it lacks the optional multiplier.

(A) (2 points) An instruction that swaps the contents of two registers, in a single clock cycle:

SWAPR(Rx, Ry) // Swap register contents
TMP ← Reg[Rx]
Reg[Rx] ← Reg[Ry]
Reg[Ry] ← TMP
PC ← PC + 4

Best implementation strategy, or None (circle one): ... macro ... new opcode ... NONE

If macro, write equivalent assembly-language instruction ; if NONE, give brief explanation:

if new opcode, fill in control signals here:

Instr	ALUFN	WERF	BSEL	WDSEL	WR	RA2SEL	PCSEL	ASEL	WASEL
SWAPR									

Problem 3 (continued):

(B) (2 points) An instruction that negates the two's-complement integer in Rx:

NEG(Rx, Ry) // two's complement negate
Reg[Ry] ← - Reg[Rx]
PC ← PC + 4

Best implementation strategy, or None (circle one): ... macro ... new opcode ... NONE

If macro, write equivalent assembly-language instruction ; if NONE, give brief explanation:

if new opcode, fill in control signals here:

Instr	ALUFN	WERF	BSEL	WDSEL	WR	RA2SEL	PCSEL	ASEL	WASEL
NEG									

(C) (2 points) A PC-relative Store instruction:

STR(Rx, C) // Store relative
EA ← PC+4+4*SEXT(C)
Mem[EA] ← Reg[Rx]
PC ← PC + 4

Best implementation strategy, or None (circle one): ... macro ... new opcode ... NONE

If macro, write equivalent assembly-language instruction ; if NONE, give brief explanation:

if new opcode, fill in control signals here:

Instr	ALUFN	WERF	BSEL	WDSEL	WR	RA2SEL	PCSEL	ASEL	WASEL
STR									

(D) (2 points) An instruction that computes bitwise $\overline{A} \bullet B$, for A in Rx and B in Ry.

BITCLR(Rx, Ry, Rz) // clear selected bits:
Reg[Rz] ← ~Reg[Rx] & Reg[Ry] // (AND Ry with **complement** of Rx)
PC ← PC + 4

Best implementation strategy, or None (circle one): ... macro ... new opcode ... NONE

If macro, write equivalent assembly-language instruction ; if NONE, give brief explanation:

if new opcode, fill in control signals here:

Instr	ALUFN	WERF	BSEL	WDSEL	WR	RA2SEL	PCSEL	ASEL	WASEL
BITCLR									

Problem 4 (7 points): **Cache Management**

Four otherwise identical Beta systems have slightly different cache configurations. Each cache has a total of 8 lines each caching a single 32-bit data word, and **caches both instruction and data fetches**. However, the caches differ in their associativity as follows:

Cache C1: Direct mapped, 8-word cache.
Cache C2: 2-way set associative (4 sets of 2 lines), LRU replacement.
Cache C3: Fully associative, LRU replacement.

Your task is to answer questions about the performance, measured by hit ratio, of these cache designs on the following tiny benchmarks. Note that each benchmark involves instruction fetches starting at location **0** and data accesses in the neighborhood of location **1024** (= 2^{10}).

```
.=0    || Benchmark B0
       CMOVE(100, R1)
LOOP:  LD(R31, 1024, R0)
       SUBC(R1, 1, R1)
       BNE(R1, LOOP)
       HALT()
```

```
.=0    || Benchmark B1
       CMOVE(100, R1)
LOOP:  LD(R31, 1024+4, R0)
       SUBC(R1, 1, R1)
       BNE(R1, LOOP)
       HALT()
```

```
.=0    || Benchmark B2
       CMOVE(100, R1)
LOOP:  LD(R31, 1024+4, R0)
       LD(R31, 1024+8, R0)
       SUBC(R1, 1, R1)
       BNE(R1, LOOP)
       HALT()
```

```
.=0    || Benchmark B3
       CMOVE(100, R1)
LOOP:  LD(R31, 1024+4, R0)
       LD(R31, 1024+8, R0)
       LD(R31, 1024+12, R0)
       SUBC(R1, 1, R1)
       BNE(R1, LOOP)
       HALT()
```

```
.=0    || Benchmark B4
       CMOVE(100, R1)
LOOP:  LD(R31, 1024+4, R0)
       LD(R31, 1024+8, R0)
       LD(R31, 1024+12, R0)
       LD(R31, 1024+16, R0)
       SUBC(R1, 1, R1)
       BNE(R1, LOOP)
       HALT()
```

(A) (1 point) Which benchmark yields the best hit ratio with cache **C1**?

(circle one): B0 … B1 … B2 … B3 … B4

(B) (2 points) Select the value that best approximates the hit ratio with cache **C1** on Benchmark **B1**.

(select approx hit ratio): 0% … 25% … 50% … 75% … 100%

(C) (2 points) Which cache yields the best hit ratio with benchmark **B3**?

(circle one): C1 … C2 … C3

(D) (2 points) Which cache, if any, yields a hit ratio of **zero** (0%) with benchmark **B4**?

(circle one): C1 … C2 … C3 … NONE

END OF QUIZ 3

1	/ 4
2	/ 11
3	/ 8
4	/ 7
	/ 30

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Fall 2010

Quiz #3: November 5, 2010

Name Q3a Solutions		*Athena login name*	*Score*
TA: Caitlin, 26-322 ☐ WF 10 ☐ WF 11	TA: Quentin, 34-303 ☐ WF 11 ☐ WF 12	TA: Sabrina, 34-304 ☐ WF 12 ☐ WF 1	TA: Steve, 34-303 ☐ WF 2

Note: There is reference material, of possible use in solving problems on this quiz, on the backs of quiz pages.

Problem 1 (4 points): **Wish I'd stayed awake during that lecture ...**

For each of the following statements, indicate whether the statement is

TRUE: proven fact; or
ACCEPTED: unproven but generally accepted as true; or
UNKNOWN: neither proven nor disproven; or
FALSE: known to be untrue.

(A) A Universal Turing Machine can compute every well-defined integer function.

Circle one: ... TRUE ... ACCEPTED ... UNKNOWN ... FALSE

(B) Every well-defined integer function that can be computed by any digital computer that could possibly be built is computable by some Turing machine.

Circle one: ... TRUE ... ACCEPTED ... UNKNOWN ... FALSE

(C) Bertrand Russell is the Pope.

Circle one: ... TRUE ... ACCEPTED ... UNKNOWN ... FALSE

(D) In mathematics, there are statements that are both well-defined and true, but which cannot be proven.

Circle one: ... TRUE ... ACCEPTED ... UNKNOWN ... FALSE

Problem 2 (11 points): **Reverse Engineering a Mystery Function**

You've been commissioned by a government agency to reverse-engineer a mysterious procedure found on the disk of a Beta system used by a cyber-terrorist cell. You've given an incomplete copy of the C source language for the function f (shown to the right), as well as its complete translation to Beta assembly code shown below:

```
// Mystery function:

int f(int x) {
  int a = x & 5;

  if (x == 0) return 0;
  else return ?????;
}
```

Note: you may wish to refer to reference material on the backs of quiz pages.

```
f:   PUSH(LP)
     PUSH(BP)
     MOVE(SP, BP)
     ALLOCATE(1)
     PUSH(R1)

     LD(BP, -12, R0)
     ANDC(R0, 5, R1)
     ST(R1, 0, BP)

xx:  BEQ(R0, bye)

     SUBC(R0, 1, R0)
     PUSH(R0)
yy:  BR(f, LP)
     DEALLOCATE(1)

     LD(BP, 0, R1)
     ADD(R1, R0, R0)

bye: POP(R1)
     MOVE(BP,SP)
     POP(BP)
     POP(LP)
     JMP(LP)
```

(A) (2 points) Give the missing expression shown in the C code as "`?????`"

a + f(x-1)

Write C expression for ????? above

(B) (1 point) Is the value of the local variable **a** stored in the stack frame of the Beta program? If so, give its offset relative to the contents of **BP**; otherwise, write "None":

Offset of a, or None: 0

(C) (3 points) Give the 32-bit binary translation of the **BR** instruction tagged **yy:**:

0 1 1 1 0 1	1 1 1 0 0	1 1 1 1 1	1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0

(fill in 1s and 0s above)

Problem 2 (continued)

```
    13C:    7
    140:    7
    144:   5C
    148:   D4
    14C:    5
    150:    3
    154:    6
    158:   A4
    15C;  14C
    160:    4
    164:    5
    168:    5
    16C:   A4
    170:  160
BP->174:    5
    178:    4
```

The function **f** is called from an external main program, and the machine is halted when a recursive call to **f** is about to execute the **BEQ** instruction tagged **xx:**. The **BP** register of the halted machine contains **0x174**, and the hex contents of a region of memory location are shown to the left.

(D) (1 point) What is the value in **SP**?

HEX contents of SP: 0x 17C

(E) (1 point) What is the value stored in the local variable **a** in the current stack frame?

HEX value of a: 0x 5

(F) (1 point) What is the address of the **BR** instruction that made the original call to **f** from the external main program?

Address of BR for original call: 0x 58

(G) (1 point) What value is currently in the **PC**?

HEX contents of PC: 0x 90

The summer intern working for you, after looking at the assembly code for f, argues that the cyber-terrorist group that wrote this code isn't very clever. He argues that one could simply delete four instructions from the assembly-language program -- an **LD**, an **ST**, an **ALLOCATE**, and a **MOVE** -- and the program would continue to work as before (but faster and using less space).

(H) (1 point) Is he right? Can one in fact delete four lines of code as described and still have a working program?

Can one optimize by deleting 4 such lines? Circle one: YES ... NO

Problem 3 (8 points): **A Better Beta**

Marketing has decided that the next model Beta needs several additional instructions, and has called you in as a consultant to decided, in each case, whether

(i) the instruction can be implemented simply as a macro, whose body contains a **single** existing Beta instruction that performs the indicated operation;

(ii) the instruction can be implemented using the existing data paths, a new opcode and appropriate control signal generation to the Beta's control ROM; or

(iii) the instruction cannot be implemented without changes to the Beta's data paths.

For each of the following proposed new instructions, you are to determine whether it can be translated (using a macro) to a single existing instruction, and, if so, to write the equivalent assembly language instruction. If it can't be translated to an existing instruction, you must determine whether it can be implemented as a new opcode using existing Beta data paths (including your ALU from Lab 3), and, if so, to specify appropriate control signals for that opcode. If neither implementation strategy will implement the indicated operation, circle NONE and give a brief (several-word) explanation.

The Beta implementation you are given is precisely that shown in lecture; its diagram, control signals, and instruction set are given for reference on the backs of pages of this quiz. Note that the ALU is the one you implemented in Lab 3, although it lacks the optional multiplier.

(A) (2 points) An instruction that swaps the contents of two registers, in a single clock cycle:

SWAPR(Rx, Ry) // Swap register contents
TMP ← Reg[Rx]
Reg[Rx] ← Reg[Ry]
Reg[Ry] ← TMP
PC ← PC + 4

Best implementation strategy, or None (circle one): ... macro ... new opcode ... NONE

If macro, write equivalent assembly-language instruction ; if NONE, give brief explanation:
Needs 2 writes to RF in single cycle

if new opcode, fill in control signals here:

Instr	ALUFN	WERF	BSEL	WDSEL	WR	RA2SEL	PCSEL	ASEL	WASEL
SWAPR									

Problem 3 (continued):

(B) (2 points) An instruction that negates the two's-complement integer in Rx:

NEG(Rx, Ry) // two's complement negate
Reg[Ry] ← - Reg[Rx]
PC ← PC + 4

Best implementation strategy, or None (circle one): ... macro ... new opcode ... NONE

If macro, write equivalent assembly-language instruction ; if NONE, give brief explanation:

SUB(R31, Rx, Ry)

if new opcode, fill in control signals here:

Instr	ALUFN	WERF	BSEL	WDSEL	WR	RA2SEL	PCSEL	ASEL	WASEL
NEG									

(C) (2 points) A PC-relative Store instruction:

STR(Rx, C) // Store relative
EA ← PC+4+4*SEXT(C)
Mem[EA] ← Reg[Rx]
PC ← PC + 4

Best implementation strategy, or None (circle one): ... macro ... new opcode ... NONE

If macro, write equivalent assembly-language instruction ; if NONE, give brief explanation:

if new opcode, fill in control signals here:

Instr	ALUFN	WERF	BSEL	WDSEL	WR	RA2SEL	PCSEL	ASEL	WASEL
STR	A	0	--	--	1	1	0	1	--

(D) (2 points) An instruction that computes bitwise $\overline{A} \cdot B$, for A in Rx and B in Ry.

BITCLR(Rx, Ry, Rz) // clear selected bits:
Reg[Rz] ← ~Reg[Rx] & Reg[Ry] // (AND Ry with **complement** of Rx)
PC ← PC + 4

Best implementation strategy, or None (circle one): ... macro ... new opcode ... NONE

If macro, write equivalent assembly-language instruction ; if NONE, give brief explanation:

if new opcode, fill in control signals here:

Instr	ALUFN	WERF	BSEL	WDSEL	WR	RA2SEL	PCSEL	ASEL	WASEL
BITCLR	$\overline{A}$ B	1	0	1	0	0	0	0	0

Problem 2 (7 points): **Cache Management**

Four otherwise identical Beta systems have slightly different cache configurations. Each cache has a total of 8 lines each caching a single 32-bit data word, and caches both instruction and data fetches. However, the caches differ in their associativity as follows:

Cache C1: Direct mapped, 8-word cache.
Cache C2: 2-way set associative (4 sets of 2 lines), LRU replacement.
Cache C3: Fully associative, LRU replacement.

Your task is to answer questions about the performance, measured by hit ratio, of these cache designs on the following tiny benchmarks. Note that each benchmark involves instruction fetches starting at location **0** and data accesses in the neighborhood of location **1024** (= 2^{10}).

```
.=0   || Benchmark B0
      CMOVE(100, R1)
LOOP: LD(R31, 1024, R0)
      SUBC(R1, 1, R1)
      BNE(R1, LOOP)
      HALT()
```

```
.=0   || Benchmark B1
      CMOVE(100, R1)
LOOP: LD(R31, 1024+4, R0)
      SUBC(R1, 1, R1)
      BNE(R1, LOOP)
      HALT()
```

```
.=0   || Benchmark B2
      CMOVE(100, R1)
LOOP: LD(R31, 1024+4, R0)
      LD(R31, 1024+8, R0)
      SUBC(R1, 1, R1)
      BNE(R1, LOOP)
      HALT()
```

```
.=0   || Benchmark B3
      CMOVE(100, R1)
LOOP: LD(R31, 1024+4, R0)
      LD(R31, 1024+8, R0)
      LD(R31, 1024+12, R0)
      SUBC(R1, 1, R1)
      BNE(R1, LOOP)
      HALT()
```

```
.=0   || Benchmark B4
      CMOVE(100, R1)
LOOP: LD(R31, 1024+4, R0)
      LD(R31, 1024+8, R0)
      LD(R31, 1024+12, R0)
      LD(R31, 1024+16, R0)
      SUBC(R1, 1, R1)
      BNE(R1, LOOP)
      HALT()
```

(A) (1 point) Which benchmark yields the best hit ratio with cache **C1**?

(circle one): B0 ... B1 ... B2 ... B3 ... B4

(B) (2 points) Select the value that best approximates the hit ratio with cache **C1** on Benchmark **B1**.

(select approx hit ratio): 0% ... 25% ... 50% ... 75% ... 100%

(C) (2 points) Which cache yields the best hit ratio with benchmark **B3**?

(circle one): C1 ... C2 ... C3

(D) (2 points) Which cache, if any, yields a hit ratio of **zero** (0%) with benchmark **B4**?

(circle one): C1 ... C2 ... C3 ... NONE

END OF QUIZ 3

Quiz 3 Debrief

11/4

Think I pulled it together

- understood program
- did stack detective
- was quite a puzzle - took me a while

Some quickies likely wrong

And perhaps 1 or 2 stack detective things

What was qu 2

- Oh the add instructions
- think did ok on that