# CS152
## Computer Architecture and Engineering
## Lecture 2

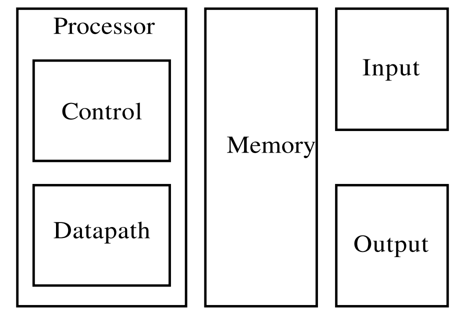
## Review of MIPS ISA and Design Concepts

January 26, 2004

John Kubiatowicz (http.cs.berkeley.edu/~kubitron)

lecture slides: http://inst.eecs.berkeley.edu/~cs152/

## Review: Organization

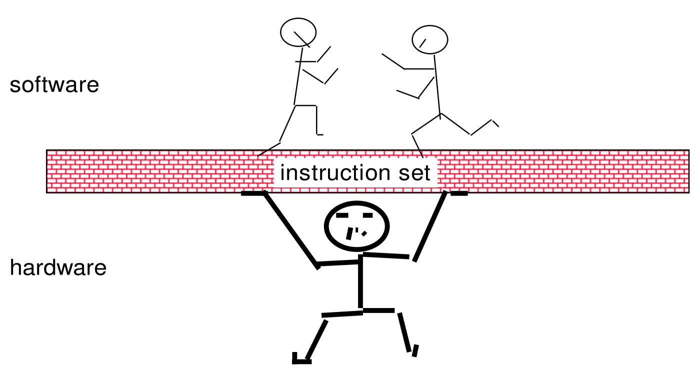



All computers consist of five components
- Processor: (1) datapath and (2) control
- (3) Memory
- I/O: (4) Input devices and (5) Output devices

Datapath and Control typically on on chip



## The Instruction Set: a Critical Interface






## Data Types

Bit: 0, 1

Bit String: sequence of bits of a particular length
- 4 bits is a nibble
- 8 bits is a byte
- 16 bits is a half-word
- 32 bits is a word
- 64 bits is a double-word

Character:
- ASCII 7 bit code
- UNICODE 16 bit code

Decimal:
- digits 0-9 encoded as 0000b thru 1001b
- two decimal digits packed per 8 bit byte

Integers:
- 2's Complement

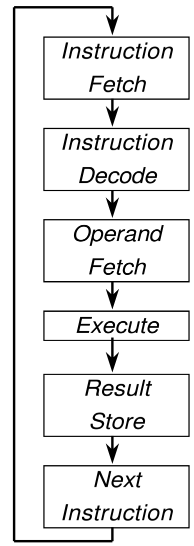
Floating Point:
- Single Precision
- Double Precision
- Extended Precision

$M \times R^E$ (mantissa, base, exponent)

How many +/- #'s?
Where is decimal pt?
How are +/- exponents represented?



## Instruction Set Architecture: What Must be Specified?




- Instruction Format or Encoding
  - how is it decoded?
- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data type and Size
- Operations
  - what are supported
- Successor instruction
  - jumps, conditions, branches
  - *fetch-decode-execute is implicit!*



## Top 10 80x86 Instructions

| Rank | instruction | Integer Average Percent total executed |
|---|---|---|
| 1 | load | 22% |
| 2 | conditional branch | 20% |
| 3 | compare | 16% |
| 4 | store | 12% |
| 5 | add | 8% |
| 6 | and | 6% |
| 7 | sub | 5% |
| 8 | move register-register | 4% |
| 9 | call | 1% |
| 10 | return | 1% |
| | Total | 96% |

Simple instructions dominate instruction frequency



## Operation Summary

Support these simple instructions, since they will dominate the number of instructions executed:

load,
store,
add,
subtract,
move register-register,
and,
shift,
compare equal, compare not equal,
branch,
jump,
call,
return;

## Methods of Testing Condition

Condition Codes

Processor status bits are set as a side-effect of arithmetic instructions (possibly on Moves) or explicitly by compare or test instructions.

ex: add r1, r2, r3
bz label

Condition Register

Ex: cmp r1, r2, r3
bgt r1, label

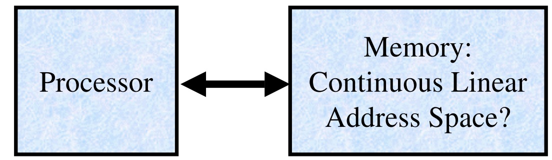
Compare and Branch

Ex: bgt r1, r2, label

Branches will be the bane of our existence in the future!



## Memory Addressing




Since 1980 almost every machine uses addresses to level of 8-bits (byte)

2 questions for design of ISA:

- How do byte addresses map onto words?
- Can a word be placed on any byte boundary?

•



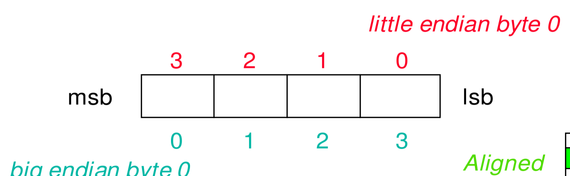
## Addressing Objects: Endianess and Alignment

Big Endian: address of most significant byte = word address (xx00 = Big End of word)

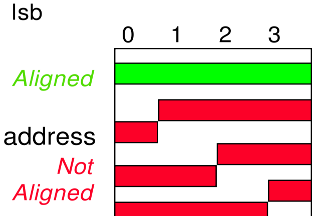
- IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

Little Endian: address of least significant byte = word address (xx00 = Little End of word)

- Intel 80x86, DEC Vax, DEC Alpha (Windows NT)




Alignment: require that objects fall on address that is multiple of their size.






## Addressing Modes

| Addressing mode | Example | Meaning |
|---|---|---|
| Register | Add R4,R3 | R4← R4+R3 |
| Immediate | Add R4,#3 | R4 ← R4+3 |
| Displacement | Add R4,100(R1) | R4 ← R4+Mem[100+R1] |
| Register indirect | Add R4,(R1) | R4 ← R4+Mem[R1] |
| Indexed / Base | Add R3,(R1+R2) | R3 ← R3+Mem[R1+R2] |
| Direct or absolute | Add R1,(1001) | R1 ← R1+Mem[1001] |
| Memory indirect | Add R1,@(R3) | R1 ← R1+Mem[Mem[R3]] |
| Post-increment | Add R1,(R2)+ | R1 ← R1+Mem[R2]; R2 ← R2+d |
| Pre-decrement | Add R1,–(R2) | R2 ← R2–d; R1 ← R1+Mem[R2] |
| Scaled | Add R1,100(R2)[R3] | R1 ← R1+Mem[100+R2+R3*d] |

*Why Post-increment/Pre-decrement? Scaled?*

## Addressing Mode Usage?

3 programs measured on machine with all address modes (VAX)

| | | |
|---|---|---|
| --- | Displacement: | 42% avg, 32% to 55% |
| --- | Immediate: | 33% avg, 17% to 43% |
| --- | Register deferred (indirect): | 13% avg, 3% to 24% |
| --- | Scaled: | 7% avg, 0% to 16% |
| --- | Memory indirect: | 3% avg, 1% to 6% |
| --- | Misc: | 2% avg, 0% to 3% |

75%

85%

75% displacement & immediate
85% displacement, immediate & register indirect

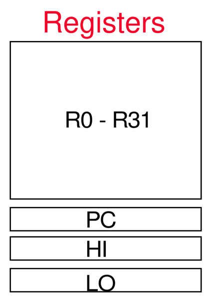





## MIPS R3000 Instruction Set Architecture (Summary)

Register Set
- 32 general 32-bit registers
- Register zero ($R0) always zero
- Hi/Lo for multiplication/division

Instruction Categories
- Load/Store
- Computational
  - Integer/Floating point
- Jump and Branch
- Memory Management
- Special



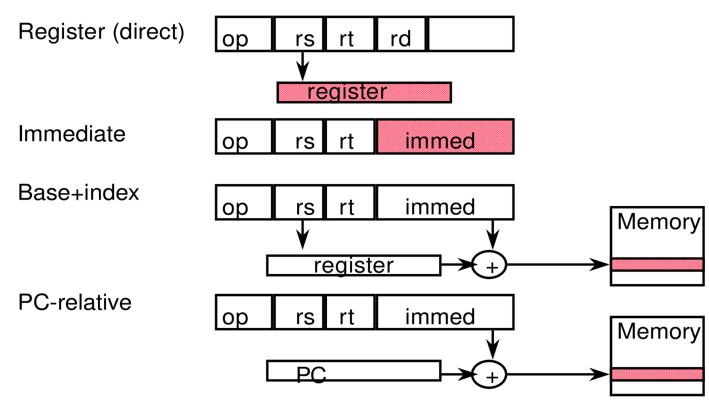

3 Instruction Formats: all 32 bits wide

| | | | | | |
|---|---|---|---|---|---|
| OP | rs | rt | rd | sa | funct |
| OP | rs | rt | immediate | | |
| OP | jump target | | | | |



## MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide




- Register Indirect?

## MIPS I Operation Overview

Arithmetic Logical:

- Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU
- AddI, AddIU, SLTI, SLTIU, AndI, OrI, XorI, LUI
- SLL, SRL, SRA, SLLV, SRLV, SRAV

Memory Access:

- LB, LBU, LH, LHU, LW, LWL,LWR
- SB, SH, SW, SWL, SWR



## Multiply / Divide

Start multiply, divide

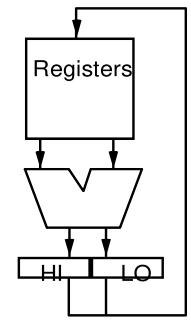
- MULT rs, rt
- MULTU rs, rt
- DIV rs, rt
- DIVU rs, rt

Move result from multiply, divide

- MFHI rd
- MFLO rd

Move to HI or LO

- MTHI rd
- MTLO rd




Why not Third field for destination?
(Hint: how many clock cycles for multiply or divide vs. add?)



## MIPS arithmetic instructions

| *Instruction* | *Example* | *Meaning* | *Comments* |
|---|---|---|---|
| add | add $1,$2,$3 | $1 = $2 + $3 | 3 operands; exception possible |
| subtract | sub $1,$2,$3 | $1 = $2 – $3 | 3 operands; exception possible |
| add immediate | addi $1,$2,100 | $1 = $2 + 100 | + constant; exception possible |
| add unsigned | addu $1,$2,$3 | $1 = $2 + $3 | 3 operands; no exceptions |
| subtract unsigned | subu $1,$2,$3 | $1 = $2 – $3 | 3 operands; no exceptions |
| add imm. unsign. | addiu $1,$2,100 | | $1 = $2 + 100 + constant; no except |
| multiply | mult $2,$3 | Hi, Lo = $2 x $3 | 64-bit signed product |
| multiply unsigned | multu$2,$3 | Hi, Lo = $2 x $3 | 64-bit unsigned product |
| divide | div $2,$3 | Lo = $2 ÷ $3, Hi = $2 mod $3 | Lo = quotient, Hi = remainder |
| divide unsigned | divu $2,$3 | Lo = $2 ÷ $3, Hi = $2 mod $3 | Unsigned quotient & remainder |
| Move from Hi | mfhi $1 | $1 = Hi | Used to get copy of Hi |
| Move from Lo | mflo $1 | $1 = Lo | Used to get copy of Lo |

Which add for address arithmetic? Which add for integers?

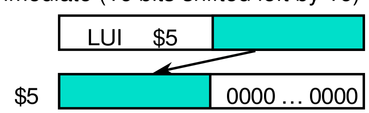


## MIPS logical instructions

| *Instruction* | *Example* | *Meaning* | *Comment* |
|---|---|---|---|
| and | and $1,$2,$3 | $1 = $2 & $3 | 3 reg. operands; Logical AND |
| or | or $1,$2,$3 | $1 = $2 \| $3 | 3 reg. operands; Logical OR |
| xor | xor $1,$2,$3 | $1 = $2 ⊕ $3 | 3 reg. operands; Logical XOR |
| nor | nor $1,$2,$3 | $1 = ~($2 \|$3) | 3 reg. operands; Logical NOR |
| and immediate | andi $1,$2,10 | $1 = $2 & 10 | Logical AND reg, constant |
| or immediate | ori $1,$2,10 | $1 = $2 \| 10 | Logical OR reg, constant |
| xor immediate | xori $1, $2,10 | $1 = ~$2 &~10 | Logical XOR reg, constant |
| shift left logical | sll $1,$2,10 | $1 = $2 << 10 | Shift left by constant |
| shift right logical | srl $1,$2,10 | $1 = $2 >> 10 | Shift right by constant |
| shift right arithm. | sra $1,$2,10 | $1 = $2 >> 10 | Shift right (sign extend) |
| shift left logical | sllv $1,$2,$3 | $1 = $2 << $3 | Shift left by variable |
| shift right logical | srlv $1,$2, $3 | $1 = $2 >> $3 | Shift right by variable |
| shift right arithm. | srav $1,$2, $3 | $1 = $2 >> $3 | Shift right arith. by variable |

## MIPS data transfer instructions

| *Instruction* | *Comment* |
|---|---|
| SW 500($4), $3 | Store word |
| SH 502($2), $3 | Store half |
| SB 41($3), $2 | Store byte |
| LW $1, 30($2) | Load word |
| LH $1, 40($3) | Load halfword |
| LHU $1, 40($3) | Load halfword unsigned |
| LB $1, 40($3) | Load byte |
| LBU $1, 40($3) | Load byte unsigned |
| LUI $1, 40 | Load Upper Immediate (16 bits shifted left by 16) |




Why need LUI?



## When does MIPS sign extend?

When value is sign extended, copy upper bit to full value:

Examples of sign extending 8 bits to 16 bits:

00001010 00000000 00001010
10001100 11111111 10001100

When is an immediate operand sign extended?

- Arithmetic instructions (add, sub, etc.) always sign extend immediates *even for the unsigned versions of the instructions!*
- Logical instructions do not sign extend immediates (They are zero extended)
- Load/Store address computations always sign extend immediates

Multiply/Divide have no immediate operands however:

- "unsigned" treat operands as unsigned

The data loaded by the instructions lb and lh are extended as follows ("unsigned" don't extend):

- lbu, lhu are zero extended
- lb, lh are sign extended



## Administrative Matters

CS152 news group: ucb.class.cs152
(email cs152@cory with specific questions)

Slides and handouts available via web:
http://inst.eecs.berkeley.edu/~cs152

Sign up to the cs152-announce mailing list:

- Link on front page of web site

Sections are on Thursday: Starting this Thursday (1/29)!

- 2:00 – 4:00 3107 Etcheverry
- 4:00 – 6:00 3107 Etcheverry

Get Cory key card/card access to Cory 119

Homework #1 due Monday at beginning of lecture

- It is up there now – really!
- Lab 1 due on following Wednesday (2/4), so get moving!

Prerequisite quiz will also be on Monday 1/29: CS 61C, CS150
Review Chapters 1-4, Ap, B of COD, Second Edition

- Look at previous versions of prerequisite quiz off handouts page

Don't forget to petition if you are on the waiting list!

- Available on third-floor, near front office
- Must turn in soon

Turn in survey forms with photo before Prereq quiz!



## MIPS Compare and Branch

Compare and Branch

- BEQ rs, rt, offset if R[rs] == R[rt] then PC-relative branch
- BNE rs, rt, offset <>

Compare to zero and Branch

- BLEZ rs, offset if R[rs] <= 0 then PC-relative branch
- BGTZ rs, offset >
- BLT <
- BGEZ >=
- BLTZAL rs, offset if R[rs] < 0 then branch and link (into R 31)
- BGEZAL >=!

Remaining set of compare and branch ops take two instructions

Almost all comparisons are against zero!

## MIPS jump, branch, compare instructions

| *Instruction* | *Example* | *Meaning* |
|---|---|---|
| branch on equal | beq $1,$2,100 | if ($1 == $2) go to PC+4+100<br>*Equal test; PC relative branch* |
| branch on not eq. | bne $1,$2,100 | if ($1!= $2) go to PC+4+100<br>*Not equal test; PC relative* |
| set on less than | slt $1,$2,$3 | if ($2 < $3) $1=1; else $1=0<br>*Compare less than; 2's comp.* |
| set less than imm. | slti $1,$2,100 | if ($2 < 100) $1=1; else $1=0<br>*Compare < constant; 2's comp.* |
| set less than uns. | sltu $1,$2,$3 | if ($2 < $3) $1=1; else $1=0<br>*Compare less than; natural numbers* |
| set l. t. imm. uns. | sltiu $1,$2,100 | if ($2 < 100) $1=1; else $1=0<br>*Compare < constant; natural numbers* |
| jump | j 10000 | go to 10000<br>*Jump to target address* |
| jump register | jr $31 | go to $31<br>*For switch, procedure return* |
| jump and link | jal 10000 | $31 = PC + 4; go to 10000<br>*For procedure call* |



## Signed vs. Unsigned Comparison

R1= **0…00 0000 0000 0000 0001**$_{two}$

R2= **0…00 0000 0000 0000 0010**$_{two}$

R3= **1…11 1111 1111 1111 1111**$_{two}$

After executing these instructions:

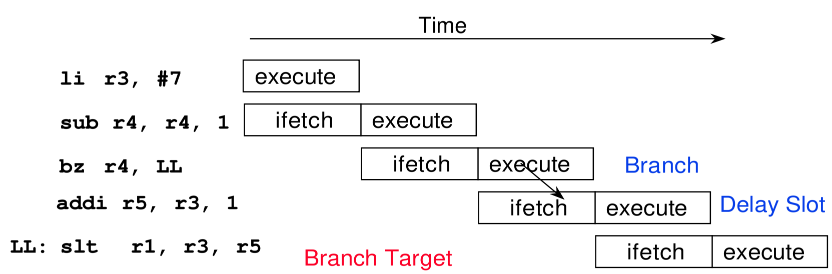
```
slt  r4,r2,r1 ;  if (r2 < r1) r4=1; else r4=0
slt  r5,r3,r1 ;  if (r3 < r1) r5=1; else r5=0
sltu r6,r2,r1 ;  if (r2 < r1) r6=1; else r6=0
sltu r7,r3,r1 ;  if (r3 < r1) r7=1; else r7=0
```

What are values of registers r4 - r7? Why?

r4 = ; r5 = ; r6 = ; r7 = ;



## Branch & Pipelines




By the end of Branch instruction, the CPU knows whether or not the branch will take place.

However, it will have fetched the next instruction by then, regardless of whether or not a branch will be taken.

Why not execute it?



## Delayed Branches

```
     li    r3, #7
     sub   r4, r4, 1
     bz    r4, LL
     addi  r5, r3, 1     ⇐ Delay Slot Instruction
     subi  r6, r6, 2
LL:  slt   r1, r3, r5
```

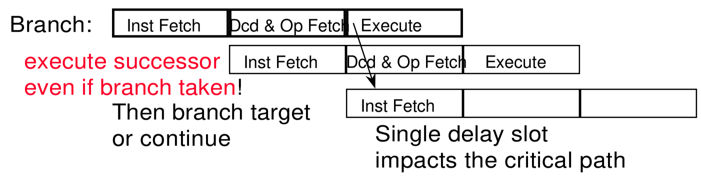
In the “Raw” MIPS, the instruction after the branch is executed even when the branch is taken

- This is hidden by the assembler for the MIPS “virtual machine”
- allows the compiler to better utilize the instruction pipeline (???)

Jump and link (jal inst):

- Put the return addr. Into link register (R31):
  - PC+4 (logical architecture)
  - PC+8 physical (“Raw”) architecture     delay slot executed
- Then jump to destination address

## Filling Delayed Branches




•Compiler can fill a single delay slot with a useful instruction 50% of the time.

- • try to move down from above jump
- •move up from target, if safe

```
        add r3, r1, r2
        sub r4, r4, 1
        bz  r4, LL
        NOP
        ...
LL:     add rd, ...
```

Is this violating the ISA abstraction?



## Miscellaneous MIPS I instructions

| | |
|---|---|
| break | A breakpoint trap occurs, transfers control to exception handler |
| syscall | A system trap occurs, transfers control to exception handler |
| coprocessor instrs. | Support for floating point |
| TLB instructions | Support for virtual memory: discussed later |
| restore from exception kernel/user | Restores previous interrupt mask & mode bits into status register |
| load word left/right | Supports misaligned word loads |
| store word left/right | Supports misaligned word stores |



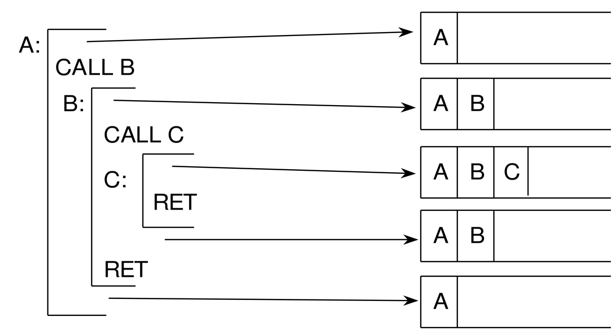
## MIPS: Software conventions for Registers

| | | | | | |
|---|---|---|---|---|---|
| 0 | zero | constant 0 | 16 | s0 | callee saves |
| 1 | at | reserved for assembler | . . . | | (callee must save) |
| 2 | v0 | expression evaluation & | 23 | s7 | |
| 3 | v1 | function results | 24 | t8 | temporary (cont'd) |
| 4 | a0 | arguments | 25 | t9 | |
| 5 | a1 | | 26 | k0 | reserved for OS kernel |
| 6 | a2 | | 27 | k1 | |
| 7 | a3 | | 28 | gp | Pointer to global area |
| 8 | t0 | temporary: caller saves | 29 | sp | Stack pointer |
| . . . | | (callee can clobber) | 30 | fp | frame pointer |
| 15 | t7 | | 31 | ra | Return Address (HW) |



## Calls: Why Are Stacks So Great?

*Stacking of Subroutine Calls & Returns and Environments:*



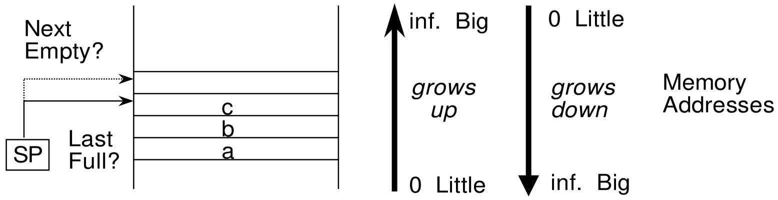

Some machines provide a memory stack as part of the architecture (e.g., VAX)

Sometimes stacks are implemented via software convention (e.g., MIPS)

## Memory Stacks

Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

*Stacks that Grow Up vs. Stacks that Grow Down:*




How is empty stack represented?

Big → Little: Last Full

POP: Read from Mem(SP)
Increment SP

PUSH: Decrement SP
Write to Mem(SP)

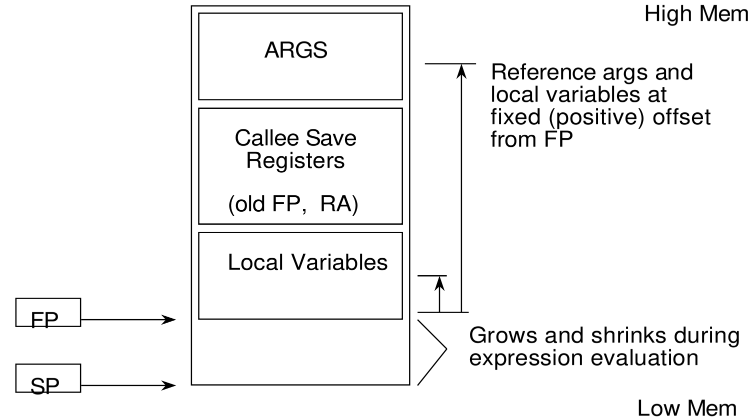
Big → Little: Next Empty

POP: Increment SP
Read from Mem(SP)

PUSH: Write to Mem(SP)
Decrement SP



## Call-Return Linkage: Stack Frames



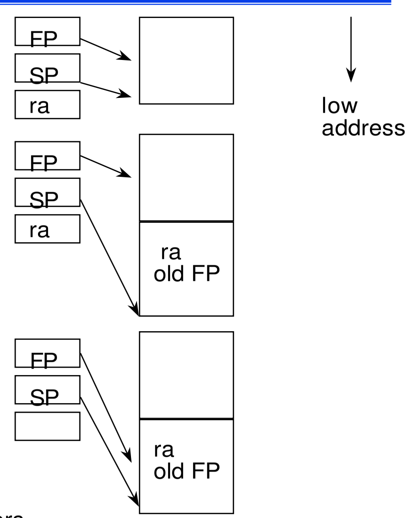

Many variations on stacks possible (up/down, last pushed / next )

Compilers normally keep scalar variables in registers, not memory!



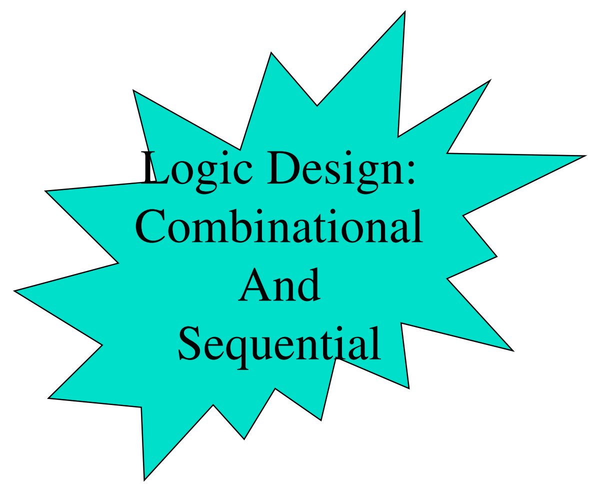
## MIPS / GCC Calling Conventions

```
fact:
  addiu    $sp, $sp, -32
  sw    $ra, 20($sp)
  sw    $fp, 16($sp)
  addiu $fp, $sp, 32
. . .
  sw    $a0, 0($fp)
...
  lw    $ra, 20($sp)
  lw    $fp, 16($sp)
  addiu $sp, $sp, 32
  jr    $ra
```




First four arguments passed in registers
Result passed in $v0/$v1





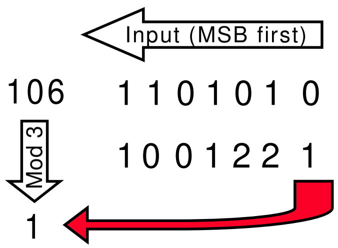
## Finite State Machines:

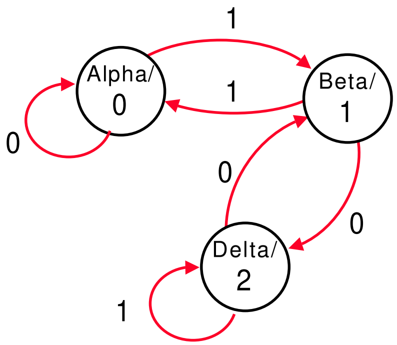
System state is explicit in representation

Transitions between states represented as arrows with inputs on arcs.

Output may be either part of state or on arcs

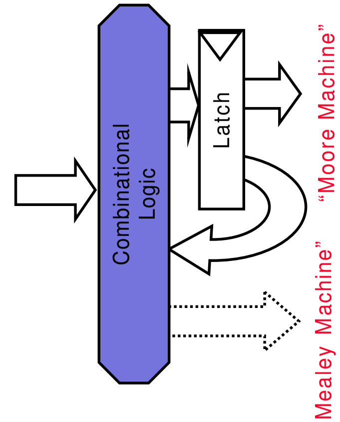
"Mod 3 Machine"

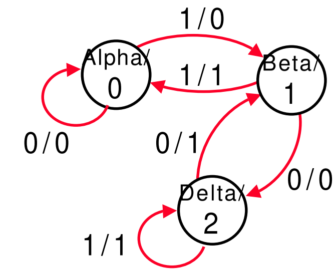





## Implementation as Combinational logic + Latch

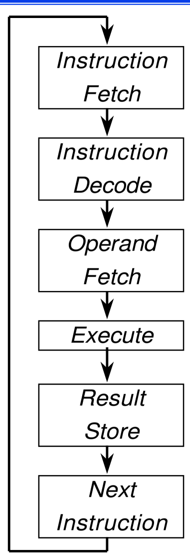



| Input | $State_{old}$ | $State_{new}$ | Div |
|---|---|---|---|
| 0 | 00 | 00 | 0 |
| 0 | 01 | 10 | 0 |
| 0 | 10 | 01 | 1 |
| 1 | 00 | 01 | 0 |
| 1 | 01 | 00 | 1 |
| 1 | 10 | 10 | 1 |



## Again: The loop of control (is there a statemachine here?)




Instruction Format or Encoding
- how is it decoded?

Location of operands and result
- where other than memory?
- how many explicit operands?
- how are memory operands located?
- which can or cannot be in memory?

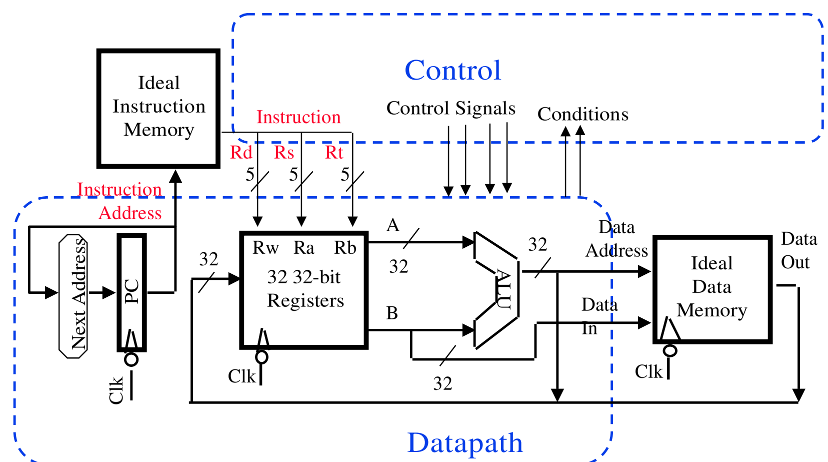
Data type and Size

Operations
- what are supported

Successor instruction
- jumps, conditions, branches
- *fetch-decode-execute is implicit!*



## Remember The Single-Cycle Datapath?

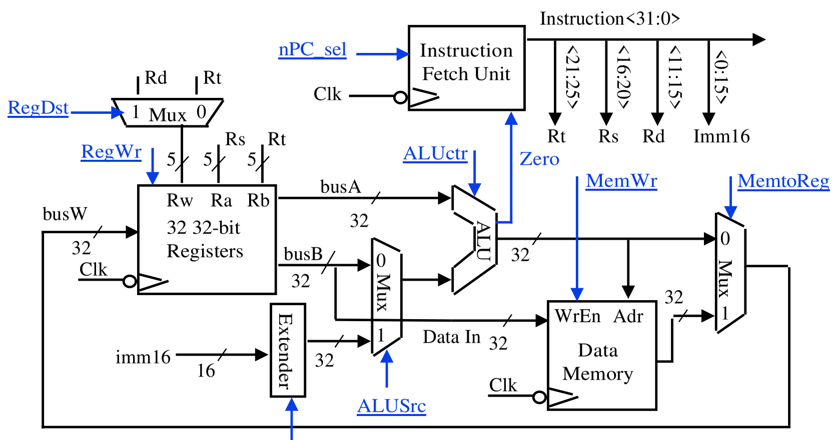



If you don't fully remember this, it is ok! (Don't need for prereq quiz)

## More detail: A Single Cycle Datapath

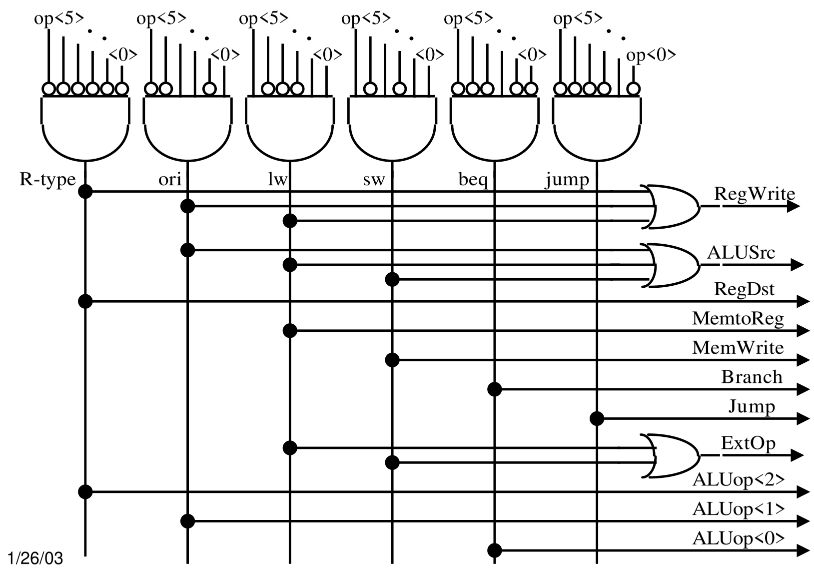
Rs, Rt, Rd and Imed16 hardwired into datapath from Fetch Unit
We have everything except ontrol signals (underline)

- Today's lecture will show you how to generate the control signals






## PLA Implementation of the Main Control



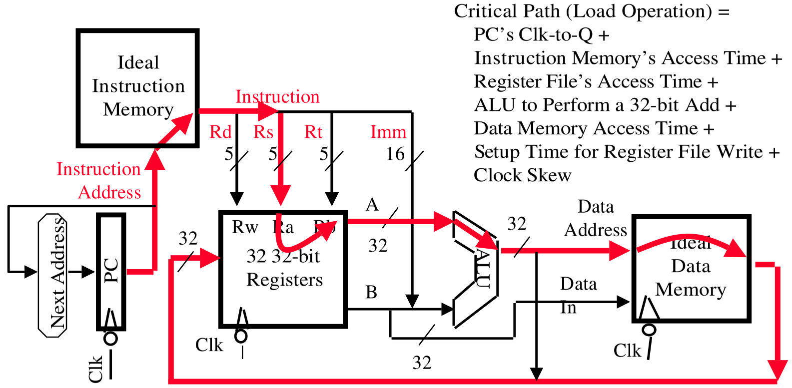



## Recap: An Abstract View of the Critical Path (Load)

Register file and ideal memory:

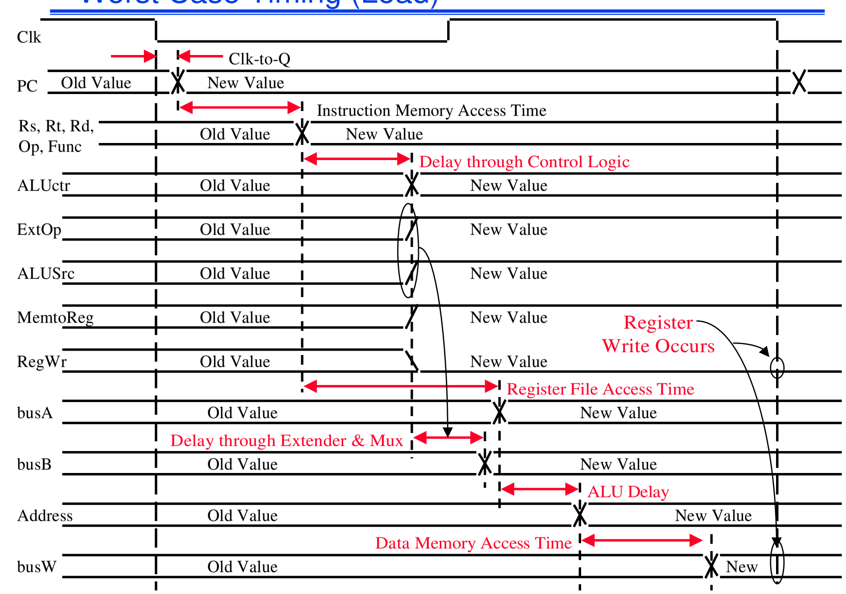
- The CLK input is a factor ONLY during write operation
- During read operation, behave as combinational logic:
  - Address valid => Output valid after "access time."

Critical Path (Load Operation) =
PC's Clk-to-Q +
Instruction Memory's Access Time +
Register File's Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

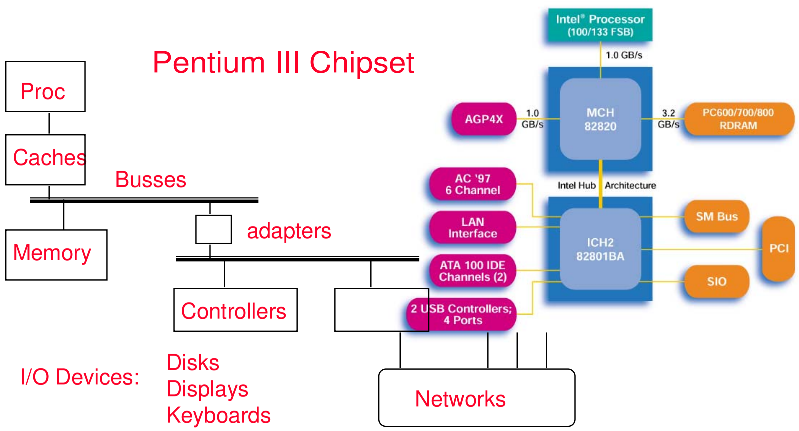





## Worst Case Timing (Load)



## Ultimately: It's all about communication




All have interfaces & organizations

Um…. It's the network stupid???!



## Summary: Salient features of MIPS I

- 32-bit fixed format inst (3 formats)
- 32 32-bit GPR (R0 contains zero) and 32 FP registers (and HI LO)
  - partitioned by software convention
- 3-address, reg-reg arithmetic instr.
- Single address mode for load/store: base+displacement
  - no indirection, scaled
- 16-bit immediate plus LUI
- Simple branch conditions
  - compare against zero or two registers for =,≠
  - no integer condition codes
- Delayed branch
  - execute instruction after a branch (or jump) even if the branch is taken

  (Compiler can fill a delayed branch with useful work about 50% of the time)