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

ISA Choices
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

Basic Format:
- M x E
- exponent
- mantissa

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:
- what are supported

Operations:
- jumps, conditions, branches
- fetch-decode-execute is implicit!

Top 10 80x86 Instructions

<table>
<thead>
<tr>
<th>Rank</th>
<th>instruction</th>
<th>Integer Average</th>
<th>Percent total executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td></td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td></td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td></td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td></td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td></td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td></td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td></td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>move register-register</td>
<td></td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td></td>
<td>1%</td>
</tr>
</tbody>
</table>

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)

Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R4,R3</td>
<td>R4 ← R4+R3</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4,#3</td>
<td>R4 ← R4+3</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4,100(R1)</td>
<td>R4 ← R4+Mem[100+R1]</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4,(R1)</td>
<td>R4 ← R4+Mem[R1]</td>
</tr>
<tr>
<td>Indexed / Base</td>
<td>Add R3,(R1+R2)</td>
<td>R3 ← R3+Mem[R1+R2]</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>R1 ← R1+Mem[1001]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1,@(R3)</td>
<td>R1 ← R1+Mem[Mem[R3]]</td>
</tr>
<tr>
<td>Post-increment</td>
<td>Add R1,(R2)+</td>
<td>R1 ← R1+Mem[R2]; R2 ← R2+d</td>
</tr>
<tr>
<td>Pre-decrement</td>
<td>Add R1,−(R2)</td>
<td>R2 ← R2−d; R1 ← R1+Mem[R2]</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>R1 ← R1+Mem[100+R2+R3*d]</td>
</tr>
</tbody>
</table>

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

MIPS I
Instruction set

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

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>imm</td>
<td></td>
</tr>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>op</td>
<td>jump target</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS Addressing Modes/Instruction Formats

• All instructions 32 bits wide

Register (direct)

Immediate

Base+index

PC-relative

• Register Indirect?
### MIPS I Operation Overview

#### Arithmetic Logical:
- Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU
- Addl, AddIU, SLLT, 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = $2 - $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = $2 + 100 + constant; exception possible</td>
<td></td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,$3</td>
<td>$1 = $2 - $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>add imm. unsign.</td>
<td>addiu $1,$2,100</td>
<td>$1 = $2 + 100 + constant; no except</td>
<td></td>
</tr>
<tr>
<td>multiply</td>
<td>mult $2,$3</td>
<td>Hi, Lo = $2 x $3 64-bit signed product</td>
<td></td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>multu$2,$3</td>
<td>Hi, Lo = $2 x $3 64-bit unsigned product</td>
<td></td>
</tr>
<tr>
<td>divide</td>
<td>div $2,$3</td>
<td>Lo = $2 ÷ $3, Hi = remainder</td>
<td>Hi = $2 mod $3</td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3, Unsigned quotient &amp; remainder</td>
<td>Hi = $2 mod $3</td>
</tr>
</tbody>
</table>

#### 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

### MIPS logical instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and $1,$2,$3</td>
<td>$1 = $2 &amp; $3</td>
<td>3 reg. operands; Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>or $1,$2,$3</td>
<td>$1 = $2</td>
<td>$3</td>
</tr>
<tr>
<td>xor</td>
<td>xor $1,$2,$3</td>
<td>$1 = $2 ⊕ $3</td>
<td>3 reg. operands; Logical XOR</td>
</tr>
<tr>
<td>nor</td>
<td>nor $1,$2,$3</td>
<td>$1 = ~(($2 $3)</td>
<td>3 reg. operands; Logical NOR</td>
</tr>
<tr>
<td>and immediate</td>
<td>andi $1,$2,10</td>
<td>$1 = $2 &amp; 10</td>
<td>Logical AND reg, constant</td>
</tr>
<tr>
<td>or immediate</td>
<td>ori $1,$2,10</td>
<td>$1 = $2</td>
<td>10</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xori $1,$2,10</td>
<td>$1 = ~(($2</td>
<td>10</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sll $1,$2,10</td>
<td>$1 = $2 &lt;&lt; 10</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srl $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift right arithmetic</td>
<td>sra $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right (sign extend)</td>
</tr>
<tr>
<td>shift left logical</td>
<td>slv $1,$2,$3</td>
<td>$1 = $2 &lt;&lt; $3</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srlv $1,$2,$3</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td>shift right arithmetic</td>
<td>srav $1,$2,$3</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right arith. by variable</td>
</tr>
</tbody>
</table>
**MIPS data transfer instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW 500($4), $3</td>
<td>Store word</td>
</tr>
<tr>
<td>SH 502($2), $3</td>
<td>Store half</td>
</tr>
<tr>
<td>SB 41($3), $2</td>
<td>Store byte</td>
</tr>
<tr>
<td>LW $1, 30($2)</td>
<td>Load word</td>
</tr>
<tr>
<td>LH $1, 40($3)</td>
<td>Load halfword</td>
</tr>
<tr>
<td>LHU $1, 40($3)</td>
<td>Load halfword unsigned</td>
</tr>
<tr>
<td>LB $1, 40($3)</td>
<td>Load byte</td>
</tr>
<tr>
<td>LBU $1, 40($3)</td>
<td>Load byte unsigned</td>
</tr>
<tr>
<td>LUI $1, 40</td>
<td>Load Upper Immediate (16 bits shifted left by 16)</td>
</tr>
</tbody>
</table>

Why need LUI?

$L5 \quad 0000 \ldots 0000$

**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

- 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch on equal</td>
<td>beq $1,$2,100</td>
<td>if ($1 == $2) go to PC+4+100</td>
</tr>
<tr>
<td></td>
<td>Equal test; PC relative</td>
<td></td>
</tr>
<tr>
<td>branch on not eq.</td>
<td>bne $1,$2,100</td>
<td>if ($1!= $2) go to PC+4+100</td>
</tr>
<tr>
<td></td>
<td>Not equal test; PC relative</td>
<td></td>
</tr>
<tr>
<td>set on less than</td>
<td>slt $1,$2,$3</td>
<td>if ($2 &lt; $3) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td>Compare less than; 2's comp.</td>
<td></td>
</tr>
<tr>
<td>set less than imm.</td>
<td>sll $1,$2,100</td>
<td>if ($2 &lt; 100) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td>Compare &lt; constant; 2's comp.</td>
<td></td>
</tr>
<tr>
<td>set l. t. imm. uns.</td>
<td>sllu $1,$2,$3</td>
<td>if ($2 &lt; 100) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td>Compare &lt; constant; natural numbers</td>
<td></td>
</tr>
<tr>
<td>jump</td>
<td>j 10000</td>
<td>go to 10000</td>
</tr>
<tr>
<td></td>
<td>Jump to target address</td>
<td></td>
</tr>
<tr>
<td>jump register</td>
<td>jr $31</td>
<td>go to $31</td>
</tr>
<tr>
<td></td>
<td>For switch, procedure return</td>
<td></td>
</tr>
<tr>
<td>jump and link</td>
<td>jal 10000</td>
<td>$31 = PC + 4; go to 10000</td>
</tr>
<tr>
<td></td>
<td>For procedure call</td>
<td></td>
</tr>
</tbody>
</table>

### Signed vs. Unsigned Comparison

R1 = 0..00 0000 0000 0000 0001<sub>two</sub>
R2 = 0..00 0000 0000 0000 0010<sub>two</sub>
R3 = 1..11 1111 1111 1111 1111<sub>two</sub>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>li r3, #7</td>
<td>execute</td>
</tr>
<tr>
<td>sub r4, r4, 1</td>
<td>ifetch</td>
</tr>
<tr>
<td></td>
<td>execute</td>
</tr>
<tr>
<td>bz r4, LL</td>
<td>ifetch</td>
</tr>
<tr>
<td></td>
<td>execute</td>
</tr>
<tr>
<td>addi r5, r3, 1</td>
<td>Branch</td>
</tr>
<tr>
<td></td>
<td>delay slot</td>
</tr>
<tr>
<td>a: slt r1, r3, r5</td>
<td>branch target</td>
</tr>
<tr>
<td></td>
<td>ifetch</td>
</tr>
<tr>
<td></td>
<td>execute</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>li r3, #7</td>
<td>execute</td>
</tr>
<tr>
<td>sub r4, r4, 1</td>
<td></td>
</tr>
<tr>
<td>bz r4, LL</td>
<td>delay slot</td>
</tr>
<tr>
<td>addi r5, r3, 1</td>
<td></td>
</tr>
<tr>
<td>subi r6, r6, 2</td>
<td></td>
</tr>
</tbody>
</table>

#### LL: slt r1, r3, r5

*Left* Delay Slot Instruction

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**

Branch:

- **Inst Fetch**
- **Dcd & Op Fetch**
- **Execute**

execute successor

even if branch taken!

Then branch target

or continue

Single delay slot

impacts the critical path

- 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
bx r4, ll
nop
...
ll: add rd, ...
```

Is this violating the ISA abstraction?

---

**MIPS: Software conventions for Registers**

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

---

**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**: Restores previous interrupt mask & kernel/user mode bits into status register
- **load word left/right**: Supports misaligned word loads
- **store word left/right**: Supports misaligned word stores

---

**Calls: Why Are Stacks So Great?**

Stacking of Subroutine Calls & Returns and Environments:

```
A: CALL B
   CALL C
   RET
   RET
B: A
   A B
   A B C
C: A
   A B
```

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:

Next Empty?          inf.  Big          0  Little
--------------------  -------           -------
SP                    grows up          grows down
Last Full?         Memory Addresses

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

High Mem
ARGS
Callee Save Registers
(old FP, RA)
Local Variables

Grows and shrinks during expression evaluation

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$f$p, $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

Logic Design:
Combinational And Sequential
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"

\[
\begin{array}{c|cc|c|c}
\text{Input (MSB first)} & 0 & 1 & 0 & 1 \text{ Mod 3} \\
0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 2 & 2 \text{ Mod 3} \\
\end{array}
\]

Implementation as Combinational logic + Latch

"Moore Machine"

<table>
<thead>
<tr>
<th>Input</th>
<th>State&lt;sub&gt;old&lt;/sub&gt;</th>
<th>State&lt;sub&gt;new&lt;/sub&gt;</th>
<th>Div</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Again: The loop of control (is there a state machine 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?

Control Signals
- Instruction
- Memory
- Data
- Address

Datapath

Data Out

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 control signals (underline)
• Today’s lecture will show you how to generate the control signals

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.”

PLA Implementation of the Main Control

Worst Case Timing (Load)
Ultimately: It's all about communication

Proc
Caches
Busses
Memory

Controllers

I/O Devices: Disks, Displays, Keyboards, Networks

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)