CSE 141: Introduction to Computer Architecture

Instruction Set Architecture (ISA)
What is Computer Architecture?

Computer Architecture =

Instruction Set Architecture

+ 

Machine Organization

How you talk to the machine

What the machine hardware looks like
An Instruction Set Architecture is an abstraction of a computational machine

- An ISA is “the agreed-upon interface between all the software that runs on the machine and the hardware that executes it.”
Computers do not speak English
And they do not speak C or Java or Python or Haskell (or...) either

“Swap two array elements.”

```c
int temp = array[index];
array[index] = array[index + 1];
array[index + 1] = temp;
```
The Instruction Set Architecture

• that part of the architecture that is visible to the programmer
  – available instructions ("opcodes")
  – number and types of registers
  – instruction formats
  – storage access, addressing modes
  – exceptional conditions
The Instruction Execution Cycle

- **Fetch**: Obtain instruction from program storage
- **Decode**: Determine required actions and instruction size
- **Operand Fetch**: Locate and obtain operand data
- **Execute**: Compute result value or status
- **Store**: Deposit results in storage for later use
- **Next Instruction**: Determine successor instruction
A brief preview of some machine organization concepts:

**Cycle**

- The smallest unit of time in a processor
A brief preview of some machine organization concepts: **Parallelism**

- The ability to do more than one thing at once

Real-world example

ARM’s Thumb instruction set is *(mostly)* 16-bit instructions on a 32-bit machine

ISA design makes fetch “freely parallel”
A brief preview of some machine organization concepts: **Superscalar Processor**

- Can execute more than one instruction per cycle

Duplication is easy but expensive...

How to do parallelism well?
- Second half of this class
- CSE148
A brief preview of some machine organization concepts: **Pipelining**

- Overlapping parts of a large task to increase throughput without decreasing latency
  - Key insight: The less work you do in one step, the faster each step can finish
Key questions to ask when designing an ISA

- **operations**
  - how many?
  - which ones?
- **operands**
  - how many?
  - location
  - types
  - how to specify?
- **instruction format**
  - size
  - how many formats?

\[
y = x + b
\]

**Syntax choice**
- \texttt{add r5, r1, r2}
- \texttt{add [r1, r2], r5}

**Design choice**
- \texttt{add r5, r1–r4}

- how does the computer know what \texttt{0001 0101 0001 0010} means?
Poll Q:
Your architecture supports 16 instructions and 16 registers (0-15). You have fixed width instructions which are 16 bits. How many register operands can you specify (explicitly) in an add instruction?

<table>
<thead>
<tr>
<th>Selection</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>&lt;= 1</td>
</tr>
<tr>
<td>B</td>
<td>&lt;= 2</td>
</tr>
<tr>
<td>C</td>
<td>&lt;= 3</td>
</tr>
<tr>
<td>D</td>
<td>&lt;= 4</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
</tr>
</tbody>
</table>
Let us design MIPS together

• We will look at several of the key ISA design decisions
• To succeed in 141 you need to understand the how and the why of MIPS
  – The rest of the course builds on MIPS, so need to be comfortable with it
  – But also need to understand the architectural tradeoffs of MIPS
• To succeed in 141L you need to understand the tradeoffs in ISA design
How long should an instruction be?

- **Fixed**

- **Variable**

- **Hybrid**

```
add r5, r1, r2
```
Instruction length tradeoffs

- **Fixed-length instructions (MIPS)**
  - easy fetch and decode
  - simplify pipelining and parallelism.
- **Variable-length instructions (Intel 80x86, VAX)**
  - multi-step fetch and decode
  - much more flexible and compact instruction set.
- **Hybrid instructions (ARM)**
  - Middle ground

⇒ **All MIPS instructions are 32 bits long.**
  - this decision impacts every other ISA decision we make because it makes instruction bits scarce.
Instruction Formats: What does each bit mean?

- Having many different instruction formats...
  - complicates decoding
  - uses more instruction bits (to specify the format)
  - Could allow us to take full advantage of a variable-length ISA

**VAX 11 instruction format**

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>1</th>
<th>n</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>A/M</td>
<td>A/M</td>
<td>A/M</td>
</tr>
</tbody>
</table>

**operand specifier**

**register disp**

- 5 r
- A r byte
- C r half word
- E r word

**index**

- 4 r m r displacement

- 8 r autoinc
# The MIPS Instruction Format

<table>
<thead>
<tr>
<th>Type</th>
<th>Format</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>R-type</td>
<td>6 bits</td>
</tr>
<tr>
<td>Immediate</td>
<td>I-type</td>
<td>6 bits</td>
</tr>
<tr>
<td>Jump</td>
<td>J-type</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- the opcode tells the machine which format
Example of instruction encoding:

<table>
<thead>
<tr>
<th></th>
<th>Register</th>
<th>Opcode</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th>Shamt</th>
<th>Funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Immediate</th>
<th>Opcode</th>
<th>Rs</th>
<th>Rt</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-type</td>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Jump</th>
<th>Opcode</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>J-type</td>
<td></td>
<td>6 bits</td>
<td></td>
</tr>
</tbody>
</table>

```
opcode=0, rs=1, rt=2, rd=5, sa=0, funct=32
000000 00001 00010 00101 00000 100000
0000000001000100010100000100000
0x00222420
```

```
add r5, r1, r2
```
Poll Q: Implications of the MIPS instruction format

<table>
<thead>
<tr>
<th>Register</th>
<th>R-type</th>
<th>Immediate</th>
<th>I-type</th>
<th>Jump</th>
<th>J-type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
<tr>
<td></td>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
</tr>
<tr>
<td></td>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td></td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td>target</td>
</tr>
</tbody>
</table>

What is the maximum number of unique operations MIPS can encode?

A: 3
B: 64
C: 127
D: 128
E: None of These
Accessing the Operands

aka, what’s allowed to go here

- Operands are generally in one of two places:
  - Registers (32 options)
  - Memory ($2^{32}$ locations)

- Registers are
  - Easy to specify
  - Close to the processor (fast access)

- The idea that we want to use registers whenever possible led to **load-store architectures**.
  - Normal arithmetic instructions only access registers
  - Only access memory with explicit loads and stores
**Poll Q: Accessing the Operands**

There are typically two locations for operands: *registers* (internal storage - $t0, $a0) and *memory*. In each column we have which (reg or mem) is better.

Which row is correct?

<table>
<thead>
<tr>
<th></th>
<th>Faster access</th>
<th>Fewer bits to specify</th>
<th>More locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Mem</td>
<td>Mem</td>
<td>Reg</td>
</tr>
<tr>
<td>B</td>
<td>Mem</td>
<td>Reg</td>
<td>Mem</td>
</tr>
<tr>
<td>C</td>
<td>Reg</td>
<td>Mem</td>
<td>Reg</td>
</tr>
<tr>
<td>D</td>
<td>Reg</td>
<td>Reg</td>
<td>Mem</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MIPS uses a load/store architecture to access operands

can do:

\[
\text{add } t0 = s1 + s2
\]

and

\[
lw t0, 32(s3)
\]

forces heavy dependence on registers, which is exactly what you want in today’s CPUs

can’t do

\[
\text{add } t0 = s1, 32(s3)
\]

– more instructions

+ fast implementation

(e.g., easy pipelining)

What pushes MIPS towards a load/store design? (hint: fixed instruction length)
How Many Operands?

*aka how many of these?*

- Most instructions have three operands (e.g., $z = x + y$).
- Well-known ISAs specify 0-3 (explicit) operands per instruction.
- Operands can be specified implicitly or explicitly.
Historically, many classes of ISAs have been explored, and trade off compactness, performance, and complexity

<table>
<thead>
<tr>
<th>Style</th>
<th># Operands</th>
<th>Example</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>0</td>
<td>add</td>
<td>$\text{tos}<em>{(N-1)} \leftarrow \text{tos}</em>{(N)} + \text{tos}_{(N-1)}$</td>
</tr>
<tr>
<td>Accumulator</td>
<td>1</td>
<td>add A</td>
<td>$\text{acc} \leftarrow \text{acc} + \text{mem}[A]$</td>
</tr>
<tr>
<td>General Purpose</td>
<td>3</td>
<td>add A B Rc</td>
<td>$\text{mem}[A] \leftarrow \text{mem}[B] + \text{Rc}$</td>
</tr>
<tr>
<td>Register</td>
<td>2</td>
<td>add A Rc</td>
<td>$\text{mem}[A] \leftarrow \text{mem}[A] + \text{Rc}$</td>
</tr>
<tr>
<td>Load/Store:</td>
<td>3</td>
<td>add Ra Rb Rc</td>
<td>$\text{Ra} \leftarrow \text{Rb} + \text{Rc}$</td>
</tr>
</tbody>
</table>
Comparing the Number of Instructions

**Code sequence for C = A + B for four classes of instruction sets:**

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Accumulator</th>
<th>GP Register</th>
<th>GP Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(register-memory)</td>
<td>(load-store)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparing the Number of Instructions

Code sequence for $C = A + B$ for four classes of instruction sets:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>GP Register</th>
<th>GP Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td></td>
<td>(register-memory)</td>
<td></td>
</tr>
<tr>
<td>Push B</td>
<td></td>
<td></td>
<td>(load-store)</td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Comparing the Number of Instructions

**Code sequence for** $C = A + B$ **for four classes of instruction sets:**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>GP Register</th>
<th>GP Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>(register-memory)</td>
<td>(load-store)</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Comparing the Number of Instructions

Code sequence for $C = A + B$ for four classes of instruction sets:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>GP Register</th>
<th>GP Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>ADD C, A, B</td>
<td></td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparing the Number of Instructions

Code sequence for $C = A + B$ for four classes of instruction sets:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>GP Register</th>
<th>GP Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>ADD C, A, B</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td></td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td></td>
<td>Add R3,R1,R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td>Store C,R3</td>
</tr>
</tbody>
</table>
Exercise: Working through alternative ISAs
[if time]
Poll Q: The destination of a MIPS add operation can be...

A. Only the top of the stack
B. Only the accumulator register
C. Any general purpose register
D. Any general purpose register or anywhere in memory
E. Any general purpose register or the top of the stack
Addressing Modes
aka: how do we specify the operand we want?

- Register direct          : R3
- Immediate (literal)      : #25
- Direct (absolute)        : M[10000]
- Register indirect        : M[R3]
- Base+Displacement        : M[R3 + 10000]
- Base+Index               : M[R3 + R4]
- Scaled Index             : M[R3 + R4*d + 10000]
- Autoincrement            : M[R3++]
- Autodecrement            : M[R3 - -]
- Memory Indirect          : M[M[R3]]
MIPS addressing modes and syntax

**register direct**

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>funct</th>
</tr>
</thead>
</table>

add $1, $2, $3

**immediate**

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

add $1, $2, #35

**base + displacement**

lw $1, disp($2)

\[ R1 = M[R2 + \text{disp}] \]

**register indirect**

\[ disp = 0 \]

\[ (rs) = 0 \]
Is this sufficient?

- measurements on the VAX show that these addressing modes (immediate, direct, register indirect, and base+displacement) represent 88% of all addressing mode usage.
- similar measurements show that 16 bits is enough for the immediate 75 to 80% of the time
- and that 16 bits is enough of a displacement 99% of the time.
- (and when these are not sufficient, it typically means we need one more instruction)
What does memory look like anyway?

• Viewed as a large, single-dimension array, with an address.
• A memory address is an index into the array
• "Byte addressing" means that the index (address) points to a byte of memory.
Memory accesses are (often) required to be “word-aligned” because of how buses and memory work

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

<table>
<thead>
<tr>
<th>Address</th>
<th>32 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

- Words are aligned
  i.e., what are the least 2 significant bits of a word address?
The MIPS ISA, so far

- fixed 32-bit instructions
- 3 instruction formats (R, I, J)
- 3-operand, load-store architecture
- 32 general-purpose registers
  - R0 always equals 0.
- 2 additional special-purpose integer registers, HI and LO, because multiply and divide produce more than 32 bits.
- registers are 32-bits wide (word)
- register, immediate, and base+displacement addressing modes
But what kinds of things do computers actually do?

• arithmetic
• logical
• data transfer
• conditional branch
• unconditional jump
Which kinds of instructions does (and doesn’t) the MIPS ISA support?

- arithmetic
  - add, subtract, multiply, divide
  - But not:

- logical
  - and, or, shift left, shift right
  - But not:

- data transfer
  - load word, store word
  - But not:
“Control Flow” describes how programs execute

- Jumps
- Procedure call (jump subroutine)
- Conditional Branch
  - Used to implement, for example, if-then-else logic, loops, etc.

- Control flow must specify two things
  - Condition under which the jump or branch is taken
  - If take, the location to read the next instruction from (“target”)
Jumps are unconditional control flow. What do they look like in MIPS?

- need to be able to jump to an absolute address sometimes
- need to be able to do procedure calls and returns

<table>
<thead>
<tr>
<th>Jump</th>
<th>J-type</th>
<th>opcode</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump</td>
<td>j</td>
<td>10000</td>
<td>PC = 10000</td>
</tr>
<tr>
<td>Jump and Link</td>
<td>jal</td>
<td>20000</td>
<td>$31 = PC + 4 and PC = 20000</td>
</tr>
</tbody>
</table>
  - used for procedure calls
| Jump register | jr    | $31    | PC = $31 |
  - used for returns, but can be useful for lots of other things
  - Q: how to encode jr instruction?

Warning: Some ISAs call jumps “unconditional branches” – useful not to for MIPS
### What is the most common use of a jal instruction and why?

<table>
<thead>
<tr>
<th>Most common use</th>
<th>Best answer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong> Procedure call</td>
<td>Jal stores the next instruction in your current function so the called function knows where to return to.</td>
</tr>
<tr>
<td><strong>B</strong> Procedure call</td>
<td>Jal enables a long jump and most procedures are a fairly long distance away</td>
</tr>
<tr>
<td><strong>C</strong> If/else</td>
<td>Jal lets you go to the if while storing pc+4 (else)</td>
</tr>
<tr>
<td><strong>D</strong> If/else</td>
<td>Jal enables a long branch and most if statements are a fairly long distance away</td>
</tr>
<tr>
<td><strong>E</strong> None of the above</td>
<td></td>
</tr>
</tbody>
</table>
What if we want to condition the control flow? Branches.

```c
do { ... ; a++; } while (a < 100);
```

- `beq` and `bne` are the only branches you need
  - `beq r1, r2, addr` => if (r1 == r2): goto addr
- But other operations can be combined...
  - `slt $1, $2, $3` => if ($2 < $3) $1 = 1; else $1 = 0
- `beq`, `bne`, `slt`, and `$zero`, can implement all fundamental conditions
  - Always, never, !=, = =, >, <=, >=, <, >(unsigned), <= (unsigned), ...

```c
if (i<j)
    w = w+1;
else
    w = 5;
```
Re-working this example

if (i < j)
  if_body:
    \( w = w + 1; \)
  else
    else_body:
      \( w = 5; \)
  after_else:

1. Need to do the comparison
   – Use “store less than”, \texttt{slt} $\$temp$, $i$, $j$
     • This writes 1 in $\$temp$ \textbf{when the condition is true}
2. Need to decide whether to branch, using only registers
   – Only have $\$zero$ available to compare with
   – The question is “should we jump over the if body”
   – Want to jump to \texttt{else_body} \textbf{when $\$temp$ is 0}
   – So we conceptually we are asking \texttt{if !(i<j)} [confusing!]
   – \texttt{beq} $\$temp$, $\$zero$, \texttt{else_body}
     – This says goto the else body \textbf{when the slt was not true}
3. Need to jump over the else body
   – Don’t do both the \texttt{if} and the \texttt{else} on accident!
   – Use “unconditional jump”
   – \texttt{j after_else}
4. Finally, fill in the bodies

\[ \texttt{slt} \$temp, \$i, \$j \]
\[ \texttt{beq} \$temp, \$zero, \texttt{else_body} \]
\[ \texttt{addi} \$w, \$w, 1 \]
\[ \texttt{j after_else} \]
\[ \texttt{addi} \$w, \$zero, 5 \]
\[ \texttt{after_else:} \]
FAQs / Extras

1. Could we have used a `bne` instead?
   - Yes, if you get the value 1 into a register
   ```
   if (i < j)
   if_body:
     w = w + 1;
   else
   else_body:
     w = 5;
   after_else:
   ```
   - But this is inefficient
     - Extra instruction
     - *Register pressure*
1. Could we have used a `bne` with no more instructions?
   – Yes... if you flip the body and “put the else first”

   ```
   if (i < j) {
       if_body:
           w = w + 1;
   } else {
       else_body:
           w = 5;
   }

   after_else:
       slt $temp, $i, $j
       bne $temp, $zero, if_body
   else_body:
       addi $w, $zero, 5
   after:
       j after
   if_body:
       addi $w, $w, 1
   after:
   ```

   – Real compilers do this sometimes
How do you specify the destination of a branch/jump?

- Unconditional jumps may go long distances
  - Function calls, returns, ...

- Studies show that almost all conditional branches go short distances from the current program counter
  - loops, if-then-else, ...

- A relative address requires (many) fewer bits than an absolute address
  - e.g., \texttt{beq $1, $2, 100} \Rightarrow \text{if ($1 == $2): } PC = (PC+4) + 100 * 4
MIPS Branch and Jump Addressing Modes

- Branches (e.g., beq) use PC-relative addressing mode
  - uses fewer bits since address typically close
  - Aka: base+displacement mode, with the PC being the base
- Jumps use pseudo-direct addressing mode
  - Recall opcode is 6 bits...
    - How many bits are available for displacement? How far can you jump?
    - 26 bits of the address is in the instruction, the rest is taken from the PC.

```
                instruction                  program counter
               --------------------------------------
                6  |  26                                     4  |  28
                                |                                         | 00
                jump destination address
```
### MIPS in one slide

#### MIPS operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>$fp, $sp, $ra, $at</td>
<td>Memory[4] ...</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

#### MIPS assembly language

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>addi $s1, $s2, 100</td>
<td>$s1 = $s2 + 100</td>
<td>Used to add constants</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Word from register to memory</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>lb $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>sb $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Byte from register to memory</td>
</tr>
<tr>
<td></td>
<td>load upper immediate</td>
<td>lui $s1, 100</td>
<td>$s1 = 100 * 2^{16}</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>beq $s1, $s2, 25</td>
<td>if ($s1 == $s2) go to PC + 4 + 100</td>
<td>Equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1, $s2, 25</td>
<td>if ($s1 != $s2) go to PC + 4 + 100</td>
<td>Not equal test; PC-relative</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than; for beq, bne</td>
</tr>
<tr>
<td></td>
<td>set less than immediate</td>
<td>slti $s1, $s2, 100</td>
<td>if ($s2 &lt; 100) $s1 = 1; else $s1 = 0</td>
<td>Compare less than constant</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>jump</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>Jump to target address</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>jr $ra</td>
<td>go to $ra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal 2500</td>
<td>$ra = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
Review — Instruction Execution in a CPU

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>0</td>
</tr>
<tr>
<td>R1</td>
<td>36</td>
</tr>
<tr>
<td>R2</td>
<td>60000</td>
</tr>
<tr>
<td>R3</td>
<td>45</td>
</tr>
<tr>
<td>R4</td>
<td>198</td>
</tr>
<tr>
<td>R5</td>
<td>12</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Program Counter

- Counter value: 20000

Instruction Buffer

Memory

- Address 20000: 10001100010000110100111000100000
- Address 20004: 00000000011000010010100000100000
- Address 80000: 00000000000000000000000000111001

CPU

- ALU
- Load/Store Unit

Memory

- Addr
- Data
Poll Q: Work an Example

- Can we figure out the code?

```c
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

<table>
<thead>
<tr>
<th>Where is k?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong> $4</td>
</tr>
<tr>
<td><strong>B</strong> $5</td>
</tr>
<tr>
<td><strong>C</strong> $15</td>
</tr>
<tr>
<td><strong>D</strong> $16</td>
</tr>
<tr>
<td><strong>E</strong> None of the above</td>
</tr>
</tbody>
</table>
MIPS ISA Tradeoffs

What if?
- 64 registers
- 20-bit immediates
- 4 operand instruction (e.g. $Y = AX + B$)
RISC Architectures

• MIPS, like SPARC, PowerPC, and Alpha AXP, is a RISC (Reduced Instruction Set Computer) ISA.
  – fixed instruction length
  – few instruction formats
  – load/store architecture

• RISC architectures worked because they enabled pipelining. They continue to thrive because they enable parallelism.
Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI (cycles per instruction)
- Sometimes referred to as “RISC vs. CISC”
  - CISC = Complex Instruction Set Computer (as alt to RISC)
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy
    instructions from 1 to 54 bytes long!
- We’ll look (briefly!) at PowerPC and 80x86
- What is ARM?
PowerPC

• Indexed addressing
  – example: `lw $t1,$a0+$s3` # $t1=Memory[$a0+$s3]
  – What do we have to do in MIPS?

• Update addressing
  – update a register as part of load (for marching through arrays)
  – example: `lwu $t0,4($s3)` # $t0=Memory[$s3+4]; $s3=$s3+4
  – What do we have to do in MIPS?

• Others:
  – load multiple/store multiple
  – a special counter register “bc Loop”
    
    ```
    decrement counter, if not 0 goto loop
    ```
80x86

1978: The Intel 8086 is announced (16 bit architecture)
1980: The 8087 floating point coprocessor is added
1982: The 80286 increases address space to 24 bits, +instructions
1985: The 80386 extends to 32 bits, new addressing modes
1989-1995: The 80486, Pentium, Pentium Pro add a few instructions
(mostly designed for higher performance)
1997: MMX is added
1999: Pentium III (same architecture)
2001: Pentium 4 (144 new multimedia instructions), simultaneous multithreading (hyperthreading)
2005: dual core Pentium processors
2006: quad core (sort of) Pentium processors
2009: Nehalem – eight-core multithreaded processors
2015: Skylake – 4-core, multithreaded, added hw security features, transactional memory...
80x86

- **Complexity:**
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    - e.g., “base or scaled index with 8 or 32 bit displacement”

- **Saving grace:**
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow
Key Points

• MIPS is a general-purpose register, load-store, fixed-instruction-length architecture.
• MIPS is optimized for fast pipelined performance, not for low instruction count.
• Historic architectures favored code size over parallelism.
• MIPS most complex addressing mode, for both branches and loads/stores is base + displacement.