Instruction Set Architecture

Jason Mars
Instruction Set Architecture

- The agreed-upon interface between all the software that runs on the machine and the hardware that executes it.
C to Assembly

C program
C to Assembly

C program

compiler

Assembly
C to Assembly

C program

compiler

Assembly

assembler

Object
C to Assembly

C program

Assembly

Object

Executable

Library

compiler

assembler

linker
C to Assembly

- C program
  \[\text{compiler}\]
  \[\text{Assembly}\]
  \[\text{assembler}\]
  \[\text{Object}\]
  \[\text{linker}\]
  \[\text{Executable}\]
  \[\text{Library}\]
  \[\text{loader}\]
  \[\text{Memory}\]
C to Assembly

- C program
  - compiler
  - Assembly
  - assembler
  - Object
  - linker
  - Library
  - Executable
  - loader
  - machine code/binary
  - Memory
Then What?
Then What?

<table>
<thead>
<tr>
<th>120007a30:</th>
<th>0f00bb27</th>
<th>ldah</th>
<th>gp,15(t12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>120007a34:</td>
<td>509cbd23</td>
<td>lda</td>
<td>gp,-25520(gp)</td>
</tr>
<tr>
<td>120007a38:</td>
<td>00005d24</td>
<td>ldah</td>
<td>t1,0(gp)</td>
</tr>
<tr>
<td>120007a3c:</td>
<td>0000bd24</td>
<td>ldah</td>
<td>t4,0(gp)</td>
</tr>
<tr>
<td>120007a40:</td>
<td>2ca422a0</td>
<td>1dl</td>
<td>t0,-23508(t1)</td>
</tr>
<tr>
<td>120007a44:</td>
<td>130020e4</td>
<td>beq</td>
<td>t0,120007a94</td>
</tr>
<tr>
<td>120007a48:</td>
<td>00003d24</td>
<td>ldah</td>
<td>t0,0(gp)</td>
</tr>
<tr>
<td>120007a4c:</td>
<td>2ca4e2b3</td>
<td>stl</td>
<td>zero,-23508(t1)</td>
</tr>
<tr>
<td>120007a50:</td>
<td>0004ff47</td>
<td>clr</td>
<td>v0</td>
</tr>
<tr>
<td>120007a54:</td>
<td>28a4e5b3</td>
<td>stl</td>
<td>zero,-23512(t4)</td>
</tr>
<tr>
<td>120007a58:</td>
<td>20a421a4</td>
<td>ldq</td>
<td>t0,-23520(t0)</td>
</tr>
<tr>
<td>120007a5c:</td>
<td>0e0020e4</td>
<td>beq</td>
<td>t0,120007a98</td>
</tr>
<tr>
<td>120007a60:</td>
<td>0204e147</td>
<td>mov</td>
<td>t0,t1</td>
</tr>
<tr>
<td>120007a64:</td>
<td>0304ff47</td>
<td>clr</td>
<td>t2</td>
</tr>
<tr>
<td>120007a68:</td>
<td>0500e0c3</td>
<td>br</td>
<td>120007a80</td>
</tr>
</tbody>
</table>
Then What?

Processor

PC

instruction memory

120007a30: 0f00bb27  ldah gp,15(t12)
120007a34: 509cbd23  lda gp,-25520(gp)
120007a38: 00005d24  ldah t1,0(gp)
120007a3c: 0000bd24  ldah t4,0(gp)
120007a40: 2ca422a0  ld1 t0,-23508(t1)
120007a44: 130020e4  beq t0,120007a94
120007a48: 00003d24  ldah t0,0(gp)
120007a4c: 2ca4e2b3  stl zero,-23508(t1)
120007a50: 0004ff47  clr v0
120007a54: 28a4e5b3  stl zero,-23512(t4)
120007a58: 20a421a4  ldq t0,-23520(t0)
120007a5c: 0e0020e4  beq t0,120007a98
120007a60: 0204e147  mov t0,t1
120007a64: 0304ff47  clr t2
120007a68: 0500e0c3  br 120007a80
Then What?

Processor
PC

instruction memory

120007a30: 0f00bb27  ldah gp,15(t12)
120007a34: 509cbd23  lda gp,-25520(gp)
120007a38: 00005d24  ldah t1,0(gp)
120007a3c: 0000bd24  ldah t4,0(gp)
120007a40: 2ca422a0  ldl t0,-23508(t1)
120007a44: 130020e4  beq t0,120007a94
120007a48: 00003d24  ldah t0,0(gp)
120007a4c: 2ca4e2b3  stl zero,-23508(t1)
120007a50: 0004ff47  clr v0
120007a54: 28a4e5b3  stl zero,-23512(t4)
120007a58: 20a421a4  ldq t0,-23520(t0)
120007a5c: 0e0020e4  beq t0,120007a98
120007a60: 0204e147  mov t0,t1
120007a64: 0304ff47  clr t2
120007a68: 0500e0c3  br 120007a80
## Then What?

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th></th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>120007a30</td>
<td>0f00bb27 ldah gp,15(t12)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120007a34</td>
<td>509cbd23 lda gp,-25520(gp)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120007a38</td>
<td>00005d24 ldah t1,0(gp)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120007a3c</td>
<td>0000bd24 ldah t4,0(gp)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120007a40</td>
<td>2ca422a0 ld1 t0,-23508(t1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120007a44</td>
<td>130020e4 beq t0,120007a94</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120007a48</td>
<td>00003d24 ldah t0,0(gp)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120007a4c</td>
<td>2ca4e2b3 stl zero,-23508(t1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120007a50</td>
<td>0004ff47 clr v0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120007a54</td>
<td>28a4e5b3 stl zero,-23512(t4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120007a58</td>
<td>20a421a4 ldq t0,-23520(t0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120007a5c</td>
<td>0e0020e4 beq t0,120007a98</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120007a60</td>
<td>0204e147 mov t0,t1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120007a64</td>
<td>0304ff47 clr t2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120007a68</td>
<td>0500e0c3 br 120007a80</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What Must We Put in an Instruction?

- An instruction must drive the Execution Cycle
What Must We Put in an Instruction?

- An instruction must drive the Execution Cycle

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Fetch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Obtain instruction from program storage</td>
</tr>
</tbody>
</table>
What Must We Put in an Instruction?

- An instruction must drive the Execution Cycle

```
  Instruction
  Fetch
  Instruction
  Decode
```

- Obtain instruction from program storage
- Determine required actions and instruction size
What Must We Put in an Instruction?

• An instruction must drive the Execution Cycle

- Instruction Fetch
- Instruction Decode
- Operand Fetch

Obtain instruction from program storage
Determine required actions and instruction size
Locate and obtain operand data
What Must We Put in an Instruction?

• An instruction must drive the Execution Cycle

![Execution Cycle Diagram]

- **Instruction Fetch**
  - Obtain instruction from program storage

- **Instruction Decode**
  - Determine required actions and instruction size

- **Operand Fetch**
  - Locate and obtain operand data

- **Execute**
  - Compute result value or status
What Must We Put in an Instruction?

- An instruction must drive the Execution Cycle

```
Instruction
  Fetch

Instruction
  Decode

Operand
  Fetch

Execute

Result
  Store
```

- Obtain instruction from program storage
- Determine required actions and instruction size
- Locate and obtain operand data
- Compute result value or status
- Deposit results in storage for later use
What Must We Put in an Instruction?

- An instruction must drive the Execution Cycle

```
Instruction
  Fetch

Instruction
  Decode

Operand
  Fetch

Execute

Result
  Store

Next
  Instruction
```

- Obtain instruction from program storage
- Determine required actions and instruction size
- Locate and obtain operand data
- Compute result value or status
- Deposit results in storage for later use
- Determine successor instruction
What Must We Put in an Instruction?

- An instruction must drive the Execution Cycle

- Instruction
  - 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

- Result
  - Store: Deposit results in storage for later use

- Next
  - Instruction: Determine successor instruction
Constructing an ISA: Key Decisions

\[ y = x + b \]

(source operands) (operation) (destination operand)

(add r1, r2, r5)
Constructing an ISA: Key Decisions

• Operations
  • How many?
  • Which ones

\[
y = x + b
\]

(operation)

destination operand

source operands

(\text{add} \ r1, r2, r5)
Constructing an ISA: Key Decisions

- Operations
  - How many?
  - Which ones
- Operands
  - How many?
  - Location
  - Types
  - How to specify?

\[ y = x + b \]

(add r1, r2, r5)
Constructing an ISA: Key Decisions

- Operations
  - How many?
  - Which ones
- Operands
  - How many?
  - Location
  - Types
  - How to specify?
- Instruction format
  - Size
  - How many formats?

\[
y = x + b
\]

\{(add \ r1, r2, r5)\}

how does the computer know what
0001 0100 1101 1111
means?
Crafting an ISA

• We’ll look at some of the decisions facing an instruction set architect, and

• how those decisions were made in the design of the MIPS instruction set.

• (Will be important as you create your own ISA ;-))
Instruction Length
Instruction Length

Fixed Length

32 bits

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shift</th>
<th>funct</th>
</tr>
</thead>
</table>

32 bits

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</th>
</tr>
</thead>
</table>

32 bits

<table>
<thead>
<tr>
<th>opcode</th>
<th>target</th>
</tr>
</thead>
</table>

32 bits

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shift</th>
<th>funct</th>
</tr>
</thead>
</table>

32 bits

<table>
<thead>
<tr>
<th>opcode</th>
<th>target</th>
</tr>
</thead>
</table>

32 bits

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</th>
</tr>
</thead>
</table>
# Instruction Length

## Fixed Length

32 bits

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shift</th>
<th>funct</th>
</tr>
</thead>
</table>

32 bits

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate / offset</th>
</tr>
</thead>
</table>

32 bits

<table>
<thead>
<tr>
<th>opcode</th>
<th>target</th>
</tr>
</thead>
</table>

32 bits

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shift</th>
<th>funct</th>
</tr>
</thead>
</table>

## Variable Length

### a. JE EIP + displacement

<table>
<thead>
<tr>
<th>JE</th>
<th>Condition</th>
<th>Displacement</th>
</tr>
</thead>
</table>

### b. CALL

<table>
<thead>
<tr>
<th>CALL</th>
<th>Offset</th>
</tr>
</thead>
</table>

### c. MOV EBX, [EDI + 45]

<table>
<thead>
<tr>
<th>MOV</th>
<th>d</th>
<th>w</th>
<th>Offset</th>
<th>Displacement</th>
</tr>
</thead>
</table>

### d. PUSH ESI

<table>
<thead>
<tr>
<th>PUSH</th>
<th>Reg</th>
</tr>
</thead>
</table>

### e. ADD EAX, #6765

<table>
<thead>
<tr>
<th>ADD</th>
<th>Reg</th>
<th>w</th>
<th>Immediate</th>
</tr>
</thead>
</table>

### f. TEST EDX, #42

<table>
<thead>
<tr>
<th>TEST</th>
<th>w</th>
<th>Postbyte</th>
<th>Immediate</th>
</tr>
</thead>
</table>
Instruction Length
Instruction Length

- Variable-length instructions (Intel 80x86, VAX) require multi-step fetch and decode, but allow for a much more flexible and compact instruction set.
Instruction Length

• Variable-length instructions (Intel 80x86, VAX) require multi-step fetch and decode, but allow for a much more flexible and compact instruction set.

• Fixed-length instructions allow easy fetch and decode, and simplify pipelining and parallelism.

• All MIPS instructions are 32 bits long.

  • this decision impacts every other ISA decision we make because it makes instruction bits scarce.
Instruction Formats
Instruction Formats

- What does each bit mean?
Instruction Formats

• What does each bit mean?
• Having many different instruction formats...
Instruction Formats

- What does each bit mean?
- Having many different instruction formats...
  - complicates decoding
Instruction Formats

- What does each bit mean?
- Having many different instruction formats...
  - complicates decoding
  - uses more instruction bits (to specify the format)
Instruction Formats

• What does each bit mean?

• Having many different instruction formats...
  • complicates decoding
  • uses more instruction bits (to specify the format)

*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**

<table>
<thead>
<tr>
<th>register disp</th>
<th>5</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>byte</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>half word</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>word</td>
<td></td>
</tr>
</tbody>
</table>

| index | 4 | r | m | r | displacement |
```
What did MIPS do?

- MIPS has 3 formats

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shift amount</td>
<td>funct</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate / offset</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6 bits</th>
<th>26 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>target</td>
</tr>
</tbody>
</table>

- The opcode tells the machine which format
  - so add r1, r2, r3 has
  - opcode=0, funct=32, rs=2, rt=3, rd=1, sa=0
  - 000000 00010 00011 00001 00000 100000
Accessing the Operands

- Operands are generally in one of two places:
  - Registers (32 int, 32 fp)
  - Memory ($2^{32}$ locations)
- Registers are
  - Easy to specify
  - Close to the processor (fast access)
- The idea that we want to access registers whenever possible led to load-store architectures. (vs register-memory e.g., x86)
  - Normal arithmetic instructions only access registers
  - Only access memory with explicit loads and stores
Load-store Architectures

- can do:
  - add r1 = r2 + r3
  - and
  - load r3, M(address)

- can’t do
  - add r1 = r2 + M(address)

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

- More instructions
- Fast implementation (e.g., easy pipelining)
- Elegant
Load-store Architectures

• can do:
  • add r1 = r2 + r3

• and
  • load r3, M(address)

• can’t do
  • add r1 = r2 + M(address)

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

• - More instructions
• + Fast implementation (e.g., easy pipelining)

• *Elegant*

Why else?
How Many Operands?

• 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.
How Many Operands - ISA Classes
How Many Operands - ISA Classes

**Accumulator:**
1 address  
add A  
acc ← acc + mem[A]
How Many Operands - ISA Classes

**Accumulator:**

1 address

add A

acc ← acc + mem[A]

**Stack:**

0 address

add
tos ← tos + next
How Many Operands - ISA Classes

**Accumulator:**
1 address  add A  acc ← acc + mem[A]

**Stack:**
0 address  add  tos ← tos + next

**General Purpose Register:**
2 address  add A B  EA(A) ← EA(A) + EA(B)
3 address  add A B C  EA(A) ← EA(B) + EA(C)
How Many Operands - ISA Classes

Accumulator:
1 address add A acc ← acc + mem[A]

Stack:
0 address add tos ← tos + next

General Purpose Register:
2 address add A B EA(A) ← EA(A) + EA(B)
3 address add A B C EA(A) ← EA(B) + EA(C)

Load/Store:
3 address add Ra Rb Rc Ra ← Rb + Rc
load Ra Rb Ra ← mem[Rb]
store Ra Rb mem[Rb] ← Ra
How Many Operands - ISA Classes

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

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>GP Register (register-memory)</th>
<th>GP Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load $A$</td>
<td>Add $B$</td>
<td>Add $R1, A$</td>
<td>Add $R2, B$</td>
</tr>
<tr>
<td>Store $C$</td>
<td></td>
<td>Add $R3, R1, R2$</td>
<td>Store $C, R3$</td>
</tr>
</tbody>
</table>
How Many Operands - ISA Classes

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>Push B</td>
<td>Add</td>
<td>Pop C</td>
</tr>
<tr>
<td></td>
<td>(register-memory)</td>
<td>(load-store)</td>
<td></td>
</tr>
</tbody>
</table>
## How Many Operands - ISA Classes

### 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 B</td>
<td>Store C</td>
</tr>
<tr>
<td>Push B</td>
<td></td>
<td></td>
<td></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>
How Many Operands - ISA Classes

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>(load-store)</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td></td>
<td>(register-memory)</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# How Many Operands - ISA Classes

Code sequence for \( C = A + B \) for four classes of instruction sets:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>GP Register (register-memory)</th>
<th>GP Register (load-store)</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>
Addressing Modes
Addressing Modes

- Register direct  R3
Addressing Modes

- Register direct   R3
- Immediate (literal)   #25
Addressing Modes

• Register direct   R3
• Immediate (literal)   #25
• Direct (absolute) M[10000]
Addressing Modes

- Register direct    R3
- Immediate (literal)    #25
- Direct (absolute) M[10000]
- Register indirect    M[R3]
Addressing Modes

- Register direct   R3
- Immediate (literal)   #25
- Direct (absolute) M[10000]

- Register indirect   M[R3]
- Base+Displacement   M[R3 + 10000]
Addressing Modes

- Register direct   R3
- Immediate (literal)  #25
- Direct (absolute) M[10000]

- Register indirect   M[R3]
- Base+Displacement M[R3 + 10000]
- Base+Index   M[R3 + R4]
Addressing Modes

• 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]
Addressing Modes

• 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++]
Addressing Modes

- 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 \( M[R3 + R4 \times d + 10000] \)
- Autoincrement \( M[R3++] \)
- Autodecrement \( M[R3 - -] \)
Addressing Modes

- 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] ]
What did MIPS do?

**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)

\[(R1 = M[R2 + disp])\]
What did MIPS do?

**register direct**

\[ \text{add }$1$, $2$, $3$ \]

**immediate**

\[ \text{add }$1$, $2$, #35 \]

**base + displacement**

\[ \text{lw }$1$, disp($2$) \]

\[ (R1 = M[R2 + disp]) \]

We get register indirect and absolute for free, but how?
What did MIPS do?

**Register Direct**

\[
\begin{array}{cccccc}
\text{OP} & \text{rs} & \text{rt} & \text{rd} & \text{sa} & \text{funct} \\
\end{array}
\]

add $1, $2, $3

**Immediate**

\[
\begin{array}{cccccc}
\text{OP} & \text{rs} & \text{rt} & \text{immediate} \\
\end{array}
\]

add $1, $2, #35

base + displacement

lw $1, disp($2)

\[(R1 = M[R2 + disp])\]

We get register indirect and absolute for free, but how?

**Register Indirect**


\[\Rightarrow \text{disp} = 0\]

**Absolute**

\[\Rightarrow (rs) = 0\]
In Previous Class...
Last Class

• ISA Design Decisions
  • Instruction Length
  • Instruction Formats
  • Addressing Modes

• What Did MIPS Do?
Instruction Length

Fixed Length

32 bits

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shift</th>
<th>amount</th>
<th>funct</th>
</tr>
</thead>
</table>

Variable Length

a. JE EIP + displacement

<table>
<thead>
<tr>
<th>JE</th>
<th>Condition</th>
<th>Displacement</th>
</tr>
</thead>
</table>

b. CALL

<table>
<thead>
<tr>
<th>CALL</th>
<th>Offset</th>
</tr>
</thead>
</table>

c. MOV EBX, [EDI + 45]

<table>
<thead>
<tr>
<th>MOV</th>
<th>d</th>
<th>w</th>
<th>postbyte</th>
<th>displacement</th>
</tr>
</thead>
</table>

d. PUSH ESI

<table>
<thead>
<tr>
<th>PUSH</th>
<th>Reg</th>
</tr>
</thead>
</table>

e. ADD EAX, #6765

<table>
<thead>
<tr>
<th>ADD</th>
<th>Reg</th>
<th>w</th>
<th>Immediate</th>
</tr>
</thead>
</table>

f. TEST EDX, #42

<table>
<thead>
<tr>
<th>TEST</th>
<th>w</th>
<th>Postbyte</th>
<th>Immediate</th>
</tr>
</thead>
</table>
**Instruction Format (What did MIPS do?)**

- MIPS has 3 formats

<table>
<thead>
<tr>
<th>R-Type</th>
<th>I-Type</th>
<th>J-Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="R-Type Diagram" /></td>
<td><img src="image2.png" alt="I-Type Diagram" /></td>
<td><img src="image3.png" alt="J-Type Diagram" /></td>
</tr>
</tbody>
</table>

- The opcode tells the machine which format
- so add r1, r2, r3 has
- opcode=0, funct=32, rs=2, rt=3, rd=1, sa=0
- 000000 00010 00011 00001 00000 100000

---

**Sunday, January 13, 13**
Addressing Modes (What did MIPS do?)

**register direct**

\[
\begin{array}{cccccc}
\text{OP} & \text{rs} & \text{rt} & \text{rd} & \text{sa} & \text{funct} \\
\end{array}
\]

\text{add} \; \$1, \; \$2, \; \$3

**immediate**

\[
\begin{array}{cccc}
\text{OP} & \text{rs} & \text{rt} & \text{immediate} \\
\end{array}
\]

\text{add} \; \$1, \; \$2, \; \#35

**base + displacement**

\[
\begin{array}{c}
\text{lw} \; \$1, \; \text{disp}($2) \\
R1 = M[R2 + \text{disp}] \\
\end{array}
\]

**register indirect**

\[
\begin{array}{c}
\Rightarrow \text{disp} = 0 \\
\Rightarrow (rs) = 0 \\
\end{array}
\]

**absolute**

\[
\begin{array}{c}
\Rightarrow (rs) = 0 \\
\end{array}
\]
Is It Sufficient

- Is MIPS good enough?
  - Limited addressing modes
  - 16 Bit limit on Immediate
  - Also affect memory operations

- 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.
btw... Memory Organization

- 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 points to a byte of memory.
Memory Organization

- 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>32 bits of data</td>
</tr>
<tr>
<td>4</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>8</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>12</td>
<td>32 bits of data</td>
</tr>
</tbody>
</table>

- \(2^{32}\) bytes with byte addresses from 0 to \(2^{32}-1\)
- \(2^{30}\) words with byte addresses 0, 4, 8, ... \(2^{32}-4\)
- Words are aligned
- i.e., what are the least 2 significant bits of a word address?
MIPS ISA thus far...

- Fixed 32-bit instructions
- 3 instruction formats
- 3-operand, load-store architecture
- 32 general-purpose registers (integer, floating point)
  - R0 always equals 0.
- 2 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
Whats Left?

• Which instructions?

• Odds and ends
Which Instructions?
Which Instructions?

• Arithmetic
Which Instructions?

• Arithmetic
  • add, subtract, multiply, divide
Which Instructions?

- Arithmetic
  - add, subtract, multiply, divide
- Logical
Which Instructions?

• Arithmetic
  • add, subtract, multiply, divide

• Logical
  • and, or, shift left, shift right
Which Instructions?

• Arithmetic
  • add, subtract, multiply, divide

• Logical
  • and, or, shift left, shift right

• Data transfer
Which Instructions?

- Arithmetic
  - add, subtract, multiply, divide
- Logical
  - and, or, shift left, shift right
- Data transfer
  - load word, store word
Which Instructions?

- Arithmetic
  - add, subtract, multiply, divide
- Logical
  - and, or, shift left, shift right
- Data transfer
  - load word, store word
- Conditional branch
Which Instructions?

• Arithmetic
  • add, subtract, multiply, divide

• Logical
  • and, or, shift left, shift right

• Data transfer
  • load word, store word

• Conditional branch
  • branch equal, branch less than, etc
Which Instructions?

- Arithmetic
  - add, subtract, multiply, divide
- Logical
  - and, or, shift left, shift right
- Data transfer
  - load word, store word
- Conditional branch
  - branch equal, branch less than, etc
- Unconditional branch (jump)
Which Instructions?

• Arithmetic
  • add, subtract, multiply, divide
• Logical
  • and, or, shift left, shift right
• Data transfer
  • load word, store word
• Conditional branch
  • branch equal, branch less than, etc
• Unconditional branch (jump)
  • jump, jump and link, return
Which Instructions?

• Arithmetic
  • add, subtract, multiply, divide

• Logical
  • and, or, shift left, shift right

• Data transfer
  • load word, store word

• Conditional branch
  • branch equal, branch less than, etc

• Unconditional branch (jump)  Control
  • jump, jump and link, return
Control Flow
Control Flow

• Conditional Branches
Control Flow

• Conditional Branches

  • if-then-else, loops, switches
Control Flow

• Conditional Branches
  
  • if-then-else, loops, switches

• Unconditional Branches (Jumps)
Control Flow

• Conditional Branches
  • if-then-else, loops, switches

• Unconditional Branches (Jumps)
  • Goto-like Jumps/Branches
Control Flow

- Conditional Branches
  - if-then-else, loops, switches

- Unconditional Branches (Jumps)
  - Goto-like Jumps/Branches
  - Function/Procedure calls
Control Flow

- Conditional Branches
  - if-then-else, loops, switches
- Unconditional Branches (Jumps)
  - Goto-like Jumps/Branches
- Function/Procedure calls

- A conditional branch must specify two things
  - Location that the branch jumps to if taken (target)
  - Condition under which the branch is taken
Specifying Branch Target
Specifying Branch Target

• Studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else). Is this a good thing? Why/Why not?
Specifying Branch Target

• Studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else). Is this a good thing? Why/Why not?

• We can specify a relative address in much fewer bits than an absolute address
Specifying Branch Target

• Studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else). Is this a good thing? Why/Why not?

• We can specify a relative address in much fewer bits than an absolute address

• e.g., beq $1, $2, 100  => if ($1 == $2) PC = (PC+4) + 100 * 4
Specifying Branch Target

• Studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else). Is this a good thing? Why/Why not?

  • We can specify a relative address in much fewer bits than an absolute address

  • e.g., beq $1, $2, 100  => if ($1 == $2) PC = (PC+4) + 100 * 4

condition
Specifying Branch’s Target

• beq, bne  
  beq r1, r2, addr => if (r1 == r2) goto addr

• slt $1, $2, $3  =>  if ($2 < $3) $1 = 1; else $1 = 0

• these, combined with $0, can implement all fundamental branch conditions

• Always, never, !=, = =, >, <=, >=, <, >(unsigned), <= (unsigned), ...

```
if (i<j)
  w = w+1;
else
  w = 5;
```
Specifying Branch’s Target

• beq, bne  
  beq r1, r2, addr => if (r1 == r2) goto addr
• slt $1, $2, $3  =>  if ($2 < $3) $1 = 1; else $1 = 0
• these, combined with $0, can implement all fundamental branch conditions
• Always, never, !=, ==, >, <=, >=, <, > (unsigned), <= (unsigned), ...

if (i<j)  
  w = w+1;  
else  
  w = 5;

slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
Specifying Branch’s Target

- beq, bne  
  \[ \text{beq } r1, r2, \text{addr } \Rightarrow \text{if } (r1 == r2) \text{ goto addr} \]
- slt $1, $2, $3  
  \[ \text{slt } $1, $2, $3 \Rightarrow \text{if } ($2 < $3) $1 = 1; \text{ else } $1 = 0 \]
- these, combined with $0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

\[
\begin{align*}
\text{if } (i < j) \\
\quad &w = w + 1; \\
\text{else} \\
\quad &w = 5;
\end{align*}
\]

\[
\begin{align*}
\text{slt } $temp, $i, $j \\
\text{beq } $temp, $0, L1 \\
\text{add } $w, $w, #1 \\
\text{beq } $0, $0, L2 \\
L1: &\text{ add } $w, $0, #5 \\
L2: &
\end{align*}
\]
Specifying Branch’s Target

- `beq, bne`  
  `beq r1, r2, addr`  
  `=> if (r1 == r2) goto addr`

- `slt $1, $2, $3`  
  `=> if ($2 < $3) $1 = 1; else $1 = 0`

- these, combined with $0, can implement all fundamental branch conditions

- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

```plaintext
if (i<j)
    w = w+1;
else
    w = 5;
```

```plaintext
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```
Specifying Branch’s Target

- `beq, bne`  
  - `beq r1, r2, addr`  
    =>  
    - if `(r1 == r2)` goto `addr`

- `slt $1, $2, $3`  
  =>  
  - if `($2 < $3)`  
    - `$1 = 1;`  
    - else `$1 = 0`

- these, combined with `$0`, can implement all fundamental branch conditions

- `Always, never, !=, = =, >, <=, >=, <, > (unsigned), <= (unsigned), ...`

```
if (i < j)
    w = w+1;
else
    w = 5;
```

```
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```
Specifying Branch’s Target

- beq, bne  
  \texttt{beq r1, r2, addr} \implies \text{if } (r1 == r2) \text{ goto addr}

- \texttt{slt $1, $2, $3} \implies \text{if } ($2 < $3) \text{ $1 = 1}; \text{ else } \text{ $1 = 0}

- these, combined with $0, can implement all fundamental branch conditions

- Always, never, !=, ==, >, <=, >=, <, >\text{(unsigned)}, <= \text{(unsigned)}, ...

\begin{verbatim}
if (i<j) 
  w = w+1;
else 
  w = 5;
\end{verbatim}

\begin{verbatim}
slt $temp, $i, $j 
beq $temp, $0, L1 
add $w, $w, #1 
beq $0, $0, L2 
L1: add $w, $0, #5 
L2:
\end{verbatim}
Specifying Branch’s Target

• beq, bne 
  beq r1, r2, addr => if (r1 == r2) goto addr

• slt $1, $2, $3 => if ($2 < $3) $1 = 1; else $1 = 0

• these, combined with $0, can implement all fundamental branch conditions

• Always, never, !=, ==, >, <=, >=, <, > (unsigned), <= (unsigned), ...

```assembly
if (i<j)
    w = w+1;
else
    w = 5;
```

```assembly
slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
```
Specifying Branch’s Target

- `beq, bne`  
  `beq r1, r2, addr`  
  `=> if (r1 == r2) goto addr`

- `slt $1, $2, $3`  
  `=> if ($2 < $3) $1 = 1; else $1 = 0`

- these, combined with $0, can implement all fundamental branch conditions

- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

if (i < j)
  w = w+1;
else
  w = 5;

\[
\text{slt } \text{temp}, \text{i}, \text{j}
\]
\[
\text{beq } \text{temp}, \text{0}, \text{L1}
\]
\[
\text{add } \text{w}, \text{w}, \text{#1}
\]
\[
\text{beq } \text{0}, \text{0}, \text{L2}
\]
\[
\text{L1: add } \text{w}, \text{0}, \text{#5}
\]
\[
\text{L2: }
\]
Specifying Branch’s Target

- `beq, bne`  
  `beq r1, r2, addr` => if (r1 == r2) goto addr

- `slt $1, $2, $3`  
  => if ($2 < $3) $1 = 1; else $1 = 0

- these, combined with $0, can implement all fundamental branch conditions

- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

\[ i=5, j=10 \]

if (i < j)
   \[ w = w+1; \]
else
   \[ w = 5; \]
Specifying Branch’s Target

• `beq, bne` 
  `beq r1, r2, addr` \( \Rightarrow \) if \((r1 == r2)\) goto addr

• `slt $1, $2, $3` 
  \( \Rightarrow \) if \((r2 < r3)\) \$1 = 1; else \$1 = 0

• these, combined with \$0, can implement all fundamental branch conditions

• Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

\[
i=5, j=10
\]

if \((i<j)\)
  \[w = w+1;\]
else
  \[w = 5;\]

\[
\begin{align*}
\text{slt } &\text{temp, } i, j \\
\text{beq } &\text{temp, } 0, \text{L1} \\
\text{add } &w, w, \#1 \\
\text{beq } &0, 0, \text{L2} \\
\text{L1: } &\text{add } w, 0, \#5 \\
\text{L2: } &
\end{align*}
\]
Specifying Branch’s Target

- `beq, bne`  
  `beq r1, r2, addr`  
  => if (r1 == r2) goto addr

- `slt $1, $2, $3`  
  => if ($2 < $3) $1 = 1; else $1 = 0

- these, combined with $0, can implement all fundamental branch conditions

- Always, never, !=, =, >, <, >=, <=, >(unsigned), <= (unsigned), ...

---

```
if (i<j)
    w = w+1;
else
    w = 5;
```

---

```
slt $temp, $i, $j
beq $temp, $0, L1
```

```
add $w, $w, #1
beq $0, $0, L2
```

```
L1: add $w, $0, #5
L2:
```
Specifying Branch’s Target

- beq, bne  
  `beq r1, r2, addr` => if \((r1 == r2)\) goto addr
- slt $1, $2, $3  
  `slt $1, $2, $3` => if \((2 < 3)\) \$1 = 1; else \$1 = 0
- these, combined with \$0, can implement all fundamental branch conditions
- Always, never, !=, = =, >, <=, >=, <, > (unsigned), <= (unsigned), ...

\[
i = 5, j = 10
\]

\[
\begin{align*}
\text{if } (i < j) \\
\quad & w = w + 1; \\
\text{else} \\
\quad & w = 5;
\end{align*}
\]

\[
\text{slt } \$\text{temp}, \$i, \$j \\
\text{beq } \$\text{temp}, \$0, \text{L1} \\
\text{add } \$w, \$w, \#1 \\
\text{beq } \$0, \$0, \text{L2} \\
\text{L1: add } \$w, \$0, \#5 \\
\text{L2:}
\]
Specifying Branch’s Target

• beq, bne  
  beq r1, r2, addr => if (r1 == r2) goto addr

• slt $1, $2, $3  =>  if ($2 < $3) $1 = 1; else $1 = 0

• these, combined with $0, can implement all fundamental branch conditions

• Always, never, !=, ==, >, <=, >=, <, > (unsigned), <= (unsigned), ...

\[
\begin{align*}
i &= 5, \quad j = 10 \\
\text{if } (i < j) &\\
\quad \text{w} &= w + 1; \quad \text{slt } \text{temp}, i, j \\
\text{else} &\\
\quad \text{w} &= 5; \\
\end{align*}
\]

\[
\begin{align*}
&\text{beq } \text{temp}, 0, \text{L1} \\
&\text{add } w, w, \#1 \\
&\text{beq } 0, 0, \text{L2} \\
&\text{L1: add } w, 0, \#5 \\
&\text{L2:}
\end{align*}
\]
Specifying Branch’s Target

- `beq, bne`  
  `beq r1, r2, addr`  
  => if \((r1 == r2)\) goto addr

- `slt $1, $2, $3`  
  => if \((\$2 < \$3)\) \$1 = 1; else \$1 = 0

- these, combined with \$0, can implement all fundamental branch conditions

- Always, never, \(!=\), \(==\), \(>\), \(<=\), \(>=\), \(<\), \(>(\text{unsigned})\), \(<=\) (\text{unsigned}), ...  

\[i=5, \ j=10\]

if \((i < j)\)

\[
\begin{align*}
\text{w} &= \text{w} + 1; \\
\text{else} & \\
\text{w} &= 5;
\end{align*}
\]

\[
\begin{align*}
\text{slt} \ \text{temp, i, j} \\
\text{beq} \ \text{temp, 0, L1} \\
\text{add} \ \text{w, w, #1} \\
\text{beq} \ 0, 0, \text{L2} \\
\text{L1: add} \ \text{w, 0, #5} \\
\text{L2:}
\end{align*}
\]
Specifying Branch’s Target

• beq, bne  
  beq r1, r2, addr  =>  if (r1 == r2) goto addr  
• slt $1, $2, $3  =>  if ($2 < $3) $1 = 1; else $1 = 0  
• these, combined with $0, can implement all fundamental branch conditions  
• Always, never, !=, = =, >, <=, >=, <, >(unsigned), <= (unsigned), ...

i=6, j=4

if (i<j)
  w = w+1;
else
  w = 5;

slt $temp, $i, $j
beq $temp, $0, L1
add $w, $w, #1
beq $0, $0, L2
L1: add $w, $0, #5
L2:
Specifying Branch’s Target

- `beq, bne`  
  `beq r1, r2, addr`  
  => if (r1 == r2) goto addr

- `slt $1, $2, $3`  
  => if ($2 < $3) $1 = 1; else $1 = 0

- these, combined with $0, can implement all fundamental branch conditions

- Always, never, !=, = =, >, <=, >=, <, >(unsigned), <= (unsigned), ...

\[
i=6, \ j=4
\]

\[
\text{if (i<j)} \\
\quad \text{w = w+1;}
\]
\[
\text{else}
\]
\[
\quad \text{w = 5;}
\]

\[
\text{slt } \text{temp}, \ i, \ j \\
\text{beq } \text{temp}, \ 0, \ L1 \\
\text{add } w, \ w, \ #1 \\
\text{beq } 0, \ 0, \ L2 \\
L1: \text{add } w, \ 0, \ #5 \\
L2:
\]
Specifying Branch’s Target

- `beq, bne`  
  `beq r1, r2, addr` => if \((r1 == r2)\) goto `addr`

- `slt $1, $2, $3` => if \((2 < 3)\) \(1 = 1\); else \(1 = 0\)

- These, combined with \(0\), can implement all fundamental branch conditions

- Always, never, \(!=\), \(=\), \(>\), \(<=\), \(>=\), \(<\), \(>\)(unsigned), \(<=\)(unsigned), ...

\[
\begin{align*}
i = 6, \; j = 4 \\
\text{if } (i < j) \\
\quad w &= w + 1; \\
\text{else} \\
\quad w &= 5;
\end{align*}
\]

\[
\begin{align*}
\text{slt } &\quad \text{$temp, i, j$} \\
\text{beq } &\quad \text{$temp, 0, L1$} \\
\text{add } &\quad \text{$w, w, 1$} \\
\text{beq } &\quad \text{$0, 0, L2$} \\
L1: &\quad \text{add $w, 0, 5$} \\
L2: &\quad \\
\end{align*}
\]
Specifying Branch’s Target

- `beq, bne`  `beq r1, r2, addr => if (r1 == r2) goto addr`
- `slt $1, $2, $3 => if ($2 < $3) $1 = 1; else $1 = 0`
- these, combined with $0, can implement all fundamental branch conditions
- Always, never, !=, ==, >, <=, >=, <, >(unsigned), <= (unsigned), ...

\[
i=6, j=4
\]

\[
\begin{align*}
\text{if (i<j)} & \quad \text{w = w+1; } \\
\text{else} & \quad \text{w = 5; }
\end{align*}
\]

\[
\begin{align*}
\text{slt } & \text{temp, i, j} \\
\text{beq } & \text{temp, 0, L1} \\
\text{add } & w, w, \#1 \\
\text{beq } & 0, 0, L2 \\
L1: & \text{add } w, 0, \#5 \\
L2: &
\end{align*}
\]
Jumps (Unconditional Branches)

- need to be able to jump to an absolute address sometime
- need to be able to do procedure calls and return

- jump -- j 10000 => PC = 10000
- jump and link -- jal 100000 => $31 = PC + 4; PC = 10000
  - used for procedure calls ($31=$ra)

- jump register -- jr $31 => PC = $31
- used for returns, but can be useful for lots of other things.
Branch and Jump Addressing Mode
Branch and Jump Addressing Mode

• Branch (e.g., beq) uses PC-relative addressing mode (uses few bits if address typically close). That is, it uses base+displacement mode, with the PC being the base. If opcode is 6 bits, how many bits are available for displacement? Determines how far can you jump.
Branch and Jump Addressing Mode

- Branch (e.g., beq) uses PC-relative addressing mode (uses few bits if address typically close). That is, it uses base+displacement mode, with the PC being the base. If opcode is 6 bits, how many bits are available for displacement? Determines how far can you jump.

- Jump uses pseudo-direct addressing mode. 26 bits of the address is in the instruction, the rest is taken from the PC.
Branch and Jump Addressing Mode

- Branch (e.g., beq) uses PC-relative addressing mode (uses few bits if address typically close). That is, it uses base+displacement mode, with the PC being the base. If opcode is 6 bits, how many bits are available for displacement? Determines how far can you jump.

- Jump uses pseudo-direct addressing mode. 26 bits of the address is in the instruction, the rest is taken from the PC.

![Diagram showing instruction and program counter with jump destination address]
### 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</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. $gp (28) is the global pointer, $sp (29) is the stack pointer, $fp (30) is the frame pointer, and $ra (31) is the return address.</td>
</tr>
<tr>
<td>$2^{30}$ memory words</td>
<td>Memory[0], Memory[4], . . . , Memory[4294967292]</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, 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 register operands</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1,$s2,$s3</td>
<td>$s1 = $s2 - $s3</td>
<td>three register operands</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>lw $s1,100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Data 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>Data from register to memory</td>
</tr>
<tr>
<td>Logical</td>
<td>and</td>
<td>and $s1,$s2,$s3</td>
<td>$s1 = $s2 &amp; $s3</td>
<td>three reg. operands; bit-by-bit AND</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>or $s1,$s2,$s3</td>
<td>$s1 = $s2</td>
<td>$s3</td>
</tr>
<tr>
<td></td>
<td>nor</td>
<td>nor $s1,$s2,$s3</td>
<td>$s1 = ~ ($s2</td>
<td>$s3)</td>
</tr>
<tr>
<td></td>
<td>and immediate</td>
<td>andi $s1,$s2,100</td>
<td>$s1 = $s2 &amp; 100</td>
<td>Bit-by-bit AND reg with constant</td>
</tr>
<tr>
<td></td>
<td>or immediate</td>
<td>ori $s1,$s2,100</td>
<td>$s1 = $s2</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>shift left logical</td>
<td>sll $s1,$s2,10</td>
<td>$s1 = $s2 &lt;&lt; 10</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td></td>
<td>shift right logical</td>
<td>srl $s1,$s2,10</td>
<td>$s1 = $s2 &gt;&gt; 10</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>beq $s1,$s2,L</td>
<td>if ($s1 == $s2) go to L</td>
<td>Equal test and branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1,$s2,L</td>
<td>if ($s1 != $s2) go to L</td>
<td>Not equal test and branch</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; used with beq, bne</td>
</tr>
<tr>
<td></td>
<td>set on less than immediate</td>
<td>slt $s1,$s2,100</td>
<td>if ($s2 &lt; 100) $s1 = 1; else $s1 = 0</td>
<td>Compare less than immediate; used with beq, bne</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>jump</td>
<td>j L</td>
<td>go to L</td>
<td>Jump to target address</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>jr $ra</td>
<td>go to $ra</td>
<td>For procedure return</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal L</td>
<td>$ra = PC + 4; go to L</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
Practice

• Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```
Practice

• Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100
Practice

- Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
$s0 = &A[0]$
$v0 = sum$
$t0 = i$
Practice

• Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
$s0 = &A[0]$
$v0 = sum;$
$t0 = i;$
Practice

• Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
$s0 = &A[0]$
$v0 = sum$
$t0 = i$
Practice

• Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
$s0 = &A[0]$
$\$v0 = sum;$
$\$t0 = i;$
Practice

• Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

Assume int is 32 bits
$s0 = &A[0]$
$v0 = sum;$
$t0 = i;$

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

```assembly
and $t0, $t0, $zero
addi $t1, $zero, 100
lw $t3, 0($s0)
```
Practice

• Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

```assembly
Assume int is 32 bits
$s0 = &A[0]
$w0 = sum;
$t0 = i;
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

```assembly
and $t0, $t0, $zero
addi $t1, $zero, 100
lw $t3, 0($s0)
add $v0, $v0, $t3
```
• Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

```assembly
Assume
int is 32 bits
$s0 = &A[0]$
$\text{v0} = \text{sum}$;
$t0 = i$;

1. initialization
2. load $A[i]$ from memory to register
3. add the value of $A[i]$ to sum
4. increase by 1
5. check if $i$ still < 100

and $t0, t0, zero$
addi $t1, zero, 100$
`lw` $t3, 0($s0$)$
add $\text{v0, v0, t3}$
addi $s0, s0, 4$
```
Practice

- Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

```assembly
and $t0, $t0, $zero
addi $t1, $zero, 100
lw $t3, 0($s0)
add $v0, $v0, $t3
addi $s0, $s0, 4
addi $t0, $t0, 1
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
$s0 = &A[0]$
$v0 = sum$
$t0 = i$
Practice

• Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

```assembly
Assume int is 32 bits
$s0 = &A[0]$  
$\text{sum} = \text{sum;\textless;100}$  
$\text{t0} = \text{i;\textless;100}$

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

and $\text{t0, t0, zero}$
addi $\text{t1, zero, 100}$
lw $\text{t3, 0(s0)}$
add $\text{v0, v0, t3}$
addi $\text{s0, s0, 4}$
addi $\text{t0, t0, 1}$
bne $\text{t1, t0, LOOP}$
```
Practice

• Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume
int is 32 bits
$s0 = &A[0]$
$v0 = sum;$
$t0 = i;$

```assembly
and $t0, $t0, $zero
addi $t1, $zero, 100
lw $t3, 0($s0)
add $v0, $v0, $t3
addi $s0, $s0, 4
addi $t0, $t0, 1
bne $t1, $t0, LOOP
```
Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

```assembly
Assume int is 32 bits
$s0 = &A[0]
$\text{v0} = \text{sum};
$\text{t0} = i;
and \ $t0, \ $t0, \ $zero
addi \ $t1, \ $zero, \ 100
lw \ $t3, \ 0($s0)
add \ $\text{v0}, \ $\text{v0}, \ $t3
addi $s0, $s0, 4
addi $t0, $t0, 1
bne $t1, $t0, LOOP

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100
```
Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

Assume

- int is 32 bits
- $s0 = \&A[0]
- $v0 = sum;
- $t0 = i;
- $t0, $t0, $zero #let i = 0
- addi $t1, $zero, 100 #temp = 100
- lw $t3, 0($s0) #temp1 = A[i]
- add $v0, $v0, $t3 #sum += temp1
- addi $s0, $s0, 4 #addr of A[i+1]
- addi $t0, $t0, 1
- bne $t1, $t0, LOOP #if i < 100

Label

```assembly
LOOP:
    lw $t3, 0($s0)
    add $v0, $v0, $t3
    addi $s0, $s0, 4
    addi $t0, $t0, 1
    bne $t1, $t0, LOOP
```
Practice

- Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

Assume
- int is 32 bits
- $s0 = &A[0]
- $v0 = sum;
- $t0 = i;

1. initialization
2. load A[i] from memory to register
3. add the value of A[i] to sum
4. increase by 1
5. check if i still < 100

```assembly
label
and $t0, $t0, $zero  #let i = 0
addi $t1, $zero, 100  #temp = 100
lw   $t3, 0($s0)  #temp1 = A[i]
add  $v0, $v0, $t3   #sum += temp1
addi $s0, $s0, 4     #addr of A[i+1]
addi $t0, $t0, 1     #i = i+1
bne  $t1, $t0, LOOP  #if i < 100
```