Chapter 3

What A CPU Does All Day

- Fetch from memory the instruction whose address is given in the PC.
- Increment the PC.
- Execute the instruction.
- Repeat.

Instructions perform operations on data in the registers, or move data between the registers and memory.

MIPS Hardware

32 Registers, Each 32 Bits

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>$0</td>
<td>The constant value 0</td>
</tr>
<tr>
<td>$at</td>
<td>$1</td>
<td>Reserved for assembler</td>
</tr>
<tr>
<td>$v0–$v1</td>
<td>$2–$3</td>
<td>Return values</td>
</tr>
<tr>
<td>$a0–$a3</td>
<td>$4–$7</td>
<td>Arguments</td>
</tr>
<tr>
<td>$t0–$t7</td>
<td>$8–$15</td>
<td>Temporaries</td>
</tr>
<tr>
<td>$s0–$s7</td>
<td>$16–$23</td>
<td>Saved</td>
</tr>
<tr>
<td>$t8–$t9</td>
<td>$24–$25</td>
<td>More temporaries</td>
</tr>
<tr>
<td>$tk0–$k1</td>
<td>$26–$27</td>
<td>Reserved for OS</td>
</tr>
<tr>
<td>$gp</td>
<td>$28</td>
<td>Global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>$29</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>$30</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>$31</td>
<td>Return address</td>
</tr>
</tbody>
</table>
### Instruction Types

Arithmetic and Logical
- Regular and immediate.
- With and without overflow.
- Bitwise logical operations.

Shift

Compare-and-Set

Jump and Branch

Load and Store

Floating Point

Described in Appendix A

---

### Arithmetic Instructions

Perform operations on 32-bit values stored in registers.

Operate on any two registers and store the result in a third.

Register names need not be unique.

The first register is the destination.

```
sub $s1, $t3, $t7
mul $s1, $s1, $t2
addu $t5, $t5, $t5
```

The $ means “unsigned.”

---

### Immediate Arithmetic Instructions

Perform operations on a 32-bit register and a 16-bit constant.

May operate on any register and store in any other. They may be the same register.

The first register is the destination.

```
addi $t7, $s5, 17
addi $t2, $10, 0x493
addi $a1, 99
```

The last one assumes $a1 is both source and destination.

---

### Logical Instructions

Take the same forms as arithmetic operations.

Perform bitwise operations.

```
0 1 0 1 0 1 0 1
and 0 1 1 0 1 1 0 0
0 0 1 0 0 1 0 0
```

```
0 1 0 1 0 1 0 1
or 0 1 1 0 1 1 0 0
```

```
1 1 1 1 1 1 1 1
```

```
0 1 0 1 0 1 0 1
or $s2, $a0, $t1
and $t2, $t1, 0x45ab
```
Shift Instructions

Shift moves the bits over to the left or right.

Bits moved past either end are discarded.

Evacuated bit positions are filled with zeros for left shifts and logical shifts.

Right arithmetic shift fills with copies of the sign bit.

Effect is to multiply or divide by a power of two, if the result fits.

- `sra $5, $t4, 3`
- `sllv $s1, $s5, $t1`

Shift Examples

These are 8-bit examples. The MIPS performs 32-bit shifts.

<table>
<thead>
<tr>
<th>Number</th>
<th>Left 3</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bin</td>
<td>dec</td>
<td>bin</td>
</tr>
<tr>
<td>00001110</td>
<td>14</td>
<td>01110000</td>
<td>112</td>
</tr>
<tr>
<td>11110100</td>
<td>-12</td>
<td>10100000</td>
<td>-96</td>
</tr>
<tr>
<td>01111001</td>
<td>121</td>
<td>11001000</td>
<td>-56</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number</th>
<th>Log. Right 2</th>
<th>Arith. Right 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bin</td>
<td>dec</td>
</tr>
<tr>
<td>00010110</td>
<td>22</td>
<td>00000101</td>
</tr>
<tr>
<td>11011000</td>
<td>-40</td>
<td>00110110</td>
</tr>
<tr>
<td>01101100</td>
<td>108</td>
<td>00011011</td>
</tr>
</tbody>
</table>

Compare-and-Set Instructions

Compare registers.

Produce a result value which is 1 or 0 (C-style boolean).

Take the same forms as arithmetic operations.

- `sgt $s2, $t6, $t5`
- `slti $t5, $a2, 10`

Jump and Branch Instructions

Move execution to another point in the program.

Jump and branch set the pc.

Branch instructions are conditional.

- `j fred`
- `beq $s1, $t1, barney`
Load and Store Instructions

Move (copy) data between register and memory.

Memory address is the sum of a register and a constant offset.

\[ \text{lw $7, 12($s1)} \]
\[ \text{sb $t3, 0($t4)} \]

Come in one-, two- and four-byte sizes.

Address must be a multiple of the size.

*Store is the only MIPS assembler instruction where data moves from left to right.*

Immediate Load Instructions

Load a constant into a register.

Do not access memory, other than the instruction itself.

\[ \text{li $a2, 123} \]
\[ \text{la $22, fred} \]

Other Instructions

Copy A Register

\[ \text{move $10, $t1} \]

Floating Point Instructions

We do not study these.

*Real Computer Scientists Don’t Do Floating Point.*

Count Down From Ten

Top:

\[ \text{li $t1, 10} \] # Set $t1 to 10.
\[ \text{beqz $t1, out} \] # Leave when not 0.
\[ \text{addi $t1, $t1, -1} \] # Decr. $t1
\[ \text{j top} \] # Repeat

Out:

\[ \ldots \]
Find Zero In An Array

... next:
la $t1, arr # Get array addr.
lw $t2, 0($t1) # Get word
beqz $t2, found # Leave when not 0.
addi $t1, $t1, 4 # Incr. address
j next # Repeat
found:
... .data .align 2
arr: .word 17, 23, 34, 11, 873, 0
...

Stack

Running programs have a stack for general storage and function calls.

- Stored in memory.
- Top denoted by $sp.
- Top is at the smallest address.
- MIPS has no special stack operations.

Stack

Grows

Lower Memory Addresses

Pushing Values

Pushing one value:
addiu $sp, $sp, -4 # Make space.
sw $s1, 0($sp) # Store value there.

Pushing several values:
addiu $sp, $sp, -12 # Make space for all.
sw $s1, 8($sp) # Store each value.
sw $s2, 4($sp)
sw $ra, 0($sp)

Note: subu $sp, $sp, $n abbreviates addiu $sp, $sp, -$n.
Popping Values

Popping one value:

```
lw   $s1, 0($sp)    # Recover value.
addiu $sp, $sp, 4  # Release space.
```

Popping several values:

```
lw   $s1, 8($sp)    # Recover each value.
lw   $s2, 4($sp)
lw   $ra, 0($sp)
addiu $sp, $sp, 12 # Release all space.
```

Calling

Call and return:

- `jal name` places address of next instruction in `$ra` and transfers to `name`.
- `jr $ra` returns by transferring to the address given in `$ra`.

```
...    fred  # Go to fred after putting
...      # this addr in $ra.
fred:   
...    # Do something useful
jr      $ra  # Go to instr after jal
```

Passing Arguments

- Arguments are passed in `$a0` through `$a3`.
- Return value(s) are left in `$v0` and `$v1`.

```
...        move $a0, ...  # Give argument.
jal fred
...      # Use $v0.
fred:     
...      # Use $a0.
move $v0, ... # Produce return value.
jr $ra
```

Use Of Registers By Functions

Caller and callee share a single set of registers.

- Caller expects `$s` registers to be preserved across a call.
- Caller may not expect other registers to be preserved.
- Callee usually saves `$s` registers it uses on the stack.

Performing a call always uses `$ra`.
- A function which calls another must preserve `$ra` for its own return.
- Functions often save `$ra` on the stack.
**Typical Function Form**

Functions generally have the following form:

```asm
funcname:
    addiu $sp, -16 # Make enough space.
    sw $ra, 0($sp)  # Save return addr.
    sw $s0, 4($sp)  # Save s regs we use.
    sw $s1, 8($sp)
    sw $s4, 12($sp) # Do whatever.

done:
    lw $s4, 12($sp) # Restore s regs.
    lw $s1, 8($sp)
    lw $s0, 4($sp)
    lw $ra, 0($sp) # Restore $ra.
    addiu $sp, 16 # Release space.
    jr $ra         # Return.
```

**Other Stack Uses**

Functions may use the stack for general storage.
- Local variables.
- Compiler temps.

Functions may wish to store $a[n] registers to free them for its own calls.

If more than four arguments are passed, they are pushed on the stack by the caller.

**Calling the Operating System**

Uses a special instruction `syscall`.

Available functions depend on OS.
*For SPIM, See p. A-48, 49.*

Printing an integer:

```asm
li $a0, 17     # Integer to print
li $v0, 1     # Code for print int
syscall        # Call OS
```

**Instruction Coding**

Instructions are coded in binary for storage in memory.

In MIPS, every instruction is represented by a 32-bit number.

Three different formats are used depending on the instruction type.
Coding Format R

- 32 bits
- | op | sreg1 | sreg2 | destreg | shift | amtsreg1 | funct |
- 6 | 5 | 5 | 5 | 5 | 6 |

```
add $10,$4,$5
0x200 4 5 10 0
0x855020
```

```
sll $8,$9,7
0x00941c0
```

```
0 0 9 8 7 0
0x000941c0
```

Coding Format I

- 32 bits
- | op | Rbase | Rdest | Offset |
- 6 | 5 | 5 | 16 |

```
lw $8, 25($11)
0x23 11 8 25
0x8d680019
```

```
beq $5,$7, fred  # fred is 104 instrs (416 bytes) ahead
4 5 7 104
0x10a70068
```

```
addi $10,$6,245
8 6 10 245
0x20ca00f5
```

Coding Format J

- 32 bits
- | op | address |
- 6 | 26 |

```
j fred  # fred is located at address 0x568ab7c
0x15a2adf
```

```
Low two bits of target address are implied; not stored
0x095a2adf
```

Multiplies and Divides

- Additional 32-bit registers hi and lo.

Multiply two 32-bit registers into 64-bit combination hi, lo.
```
mult r1, r2
```

Quotient in lo, remainder in hi.
```
div r1, r2:
mfhi r
mflo r
```
Compiler Directives

.data
.text
.ascii
.asciiiz
.space n
.word n1, ..., nk

Pseudo-Operations

move Rd, Rsrc add Rd, Rsrc, $0
neg Rd, Rsrc sub Rd,$0,Rsrc
li Rd, Small ori Rd, $0, Small
li Rd, Large lui Rd, Hi Large
    ori Rd, Rd, Lo Large

More Pseudo-Operations

mul Rd,Rs,Rt mult Rs,Rt
    mflo Rd
rol Rd,Rs,Rt subu $1, $0, Rt
    srlv $1, Rs, $1
    sl1v Rd, Rs, Rt
    or Rd, Rd, $1

Pseudo-Addressing Mode

lw       Rd, fred
    ...
fred:
    .word 100

gives

    lui     $1, hi_addr_fred
    lw      Rd, low_addr_fred($1)
    ...
fred:
    .word 100
Translating If

if (e) s
  b!e skip
s
skip:

Translating If Else

if (e) s1 else s2
  b!e epart
  s1
  j out
epart:
  s2
out:

Translating If

if(a < b) {
  a = a + b;
  b = 0;
}

bge $s0, $s1, skip
add $s0, $s0, $s1
move $s1, $zero
skip:

Translating If Else

if(a < b) {
  a = a + b;
  b = 0;
} else {
  b = b + a;
  a = 0;
}

bge $s0, $s1, epart
add $s0, $s0, $s1
move $s1, $zero
j out
epart:
  add $s1, $s1, $s0
  move $s0, $zero
out:
Translating While

while (e) s

top:
    b!e    out
    s
    j    top
out:

Translation of While

```c
while (a > 0) {
    b = b + a;
    --a;
}
```

top:
    blez $s0, out
    add $s1, $s1, $s0
    addi $s0, -1
    bgtz $s0, top
out:

Sometimes a Bottom Test will Do

```c
/* a is known positive */
while (a > 0) {
    b = b + a;
    --a;
}
```

top:
    add $s1, $s1, $s0
    addi $s0, -1
    bgtz $s0, top
out:

VAX: Addressing modes

16 Registers, including PC.
32-bit memory addresses.

Each instruction

- Two or three arguments
- Many ways to specify
VAX: Addressing modes

Ways to specify what data an instruction operates on

- Literal, \( \#v \): 6-bit constant \( v \)
- Immediate, \( \#v \): Larger constant \( v \)
- Register, \( r \): \textit{itself}
- Register Dereferenced, \((r)\): \textit{mem}[r]
- Displaced, offset(\( r \)): \textit{mem}[r + offset]

VAX: More Addressing modes

- Displaced Deferred, \( \@ \text{offset}(r) \): \textit{mem}[\textit{mem}[r + offset]]
- Indexed, base \([r]\): \textit{Add} \( r \times \text{data size to another mode.} \)
- Autoincrement, \((r)+\): \textit{mem}[r]; \( r = r + d \)
- Autodecrement, -(\( r \)): \( r = r - d; \textit{mem}[r] \)
- Autoincrement Dereferenced, \( @(r)+\): \textit{mem}[\textit{mem}[r]]; \( r = r + d \)

Offsets and immediates may by 8, 16, or 32 bits

VAX: Coding

<table>
<thead>
<tr>
<th>Op Code</th>
<th>Arg 1</th>
<th>Arg 2</th>
</tr>
</thead>
</table>

8 bits

<table>
<thead>
<tr>
<th>Op Code</th>
<th>Arg 1</th>
<th>Arg 2</th>
<th>Arg 3</th>
</tr>
</thead>
</table>

8 bits

Each arg specifies its addressing mode, and the specifications may vary in size.

VAX: Arguments

Register

<table>
<thead>
<tr>
<th>Code</th>
<th>Reg.</th>
</tr>
</thead>
</table>

16-bit displacement

<table>
<thead>
<tr>
<th>Code</th>
<th>Reg.</th>
<th>Offset</th>
</tr>
</thead>
</table>

Generally, one byte plus any displacement or immediate.
VAX and MIPS

MIPS requires about twice as many instructions.

VAX takes about three times as many clock cycles.

80x86 History

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, adds instructions.

1985: The 80386 extends to 32 bits, new addressing modes.


1997: MMX is added.

The “golden handcuffs” of compatibility.

80x86

Runs in all standard PC’s.

Variable-Length Instructions.

Register-Register, Register-Memory Instructions.

Special-Purpose Registers

Victim of Success:
Putting the “backward” in backward compatible.

80386 Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>GPR0</td>
</tr>
<tr>
<td>ECX</td>
<td>GPR1</td>
</tr>
<tr>
<td>EDX</td>
<td>GPR2</td>
</tr>
<tr>
<td>EBX</td>
<td>GPR3</td>
</tr>
<tr>
<td>ESP</td>
<td>GPR4</td>
</tr>
<tr>
<td>EBP</td>
<td>GPR5</td>
</tr>
<tr>
<td>ESI</td>
<td>GPR6</td>
</tr>
<tr>
<td>EDI</td>
<td>GPR7</td>
</tr>
<tr>
<td>CS</td>
<td>Code segment pointer</td>
</tr>
<tr>
<td>SS</td>
<td>Stack segment pointer (top of stack)</td>
</tr>
<tr>
<td>DS</td>
<td>Data segment pointer 0</td>
</tr>
<tr>
<td>ES</td>
<td>Data segment pointer 1</td>
</tr>
<tr>
<td>FS</td>
<td>Data segment pointer 2</td>
</tr>
<tr>
<td>GS</td>
<td>Data segment pointer 3</td>
</tr>
<tr>
<td>EIP</td>
<td>Instruction pointer (PC)</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>Condition codes</td>
</tr>
</tbody>
</table>
80x86 Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE name</td>
<td>If equal (CC) EIP = name; EIP - 128 ≤ name &lt; EIP + 128</td>
</tr>
<tr>
<td>JMP name</td>
<td>(EIP = NAME);</td>
</tr>
<tr>
<td>CALL name</td>
<td>SP = SP - 4; M[SP] = EIP + 5; EIP = name;</td>
</tr>
<tr>
<td>MOVW EBX,[EDI + 45]</td>
<td>EBX = M[EDI + 45]</td>
</tr>
<tr>
<td>PUSH ESI</td>
<td>SP = SP - 4; M[SP] = ESI</td>
</tr>
<tr>
<td>POP EDI</td>
<td>EDI = M[SP]; SP = SP + 4</td>
</tr>
<tr>
<td>ADD EAX,#6765</td>
<td>EAX = EAX + 6765</td>
</tr>
<tr>
<td>TEST EDX,#42</td>
<td>Set condition code (flags) with EDX &amp; 42</td>
</tr>
<tr>
<td>MOVSL</td>
<td>M[EDI] = M[ESI]; EDI = EDI + 4; ESI = ESI + 4</td>
</tr>
</tbody>
</table>

80x86 Instruction Coding Examples

a. JE EIP + displacement
   4 4 8
   CE Condition Displacement

b. CALL
   8 32
   CALL Offset

c. MOV EBX,[EDI + 45]
   6 11 8 8
   MOV [w] possible Displacement

d. PUSH ESI
   5 3
   PUSH Flag

e. ADD EAX,#6765
   4 3 1 32
   ADD Flag Immediate

f. TEST EDX,#42
   7 1 8 32
   TEST Immediate

CSc 314 - T W Bennet - Mississippi College