22C: 040

Computer Organization and Hardware

Draft Lecture Notes
Spring 2004

Sukumar Ghosh
The University of Iowa

The notes are unverified, and may contain mistakes.
Introduction

How does a computer execute a program?

Program $\rightarrow$ computer

What is there inside a computer?
Are all computer hardwares alike?
What is the difference between a PC and a Mac?
Technologies

A computer is an instruction-execution engine.

Different hardware technologies are possible:

- Mechanical
- Pneumatic
- Electronic
- Quantum
- Biological

We will focus on electronic technology only, which is most common.
Classification

General purpose

Your PC

Special purpose

The computers in your car
The computer in your cell phone
The computer inside your camera
The computer in your washing machine
History of modern day computers

Eckert and Mauchley

Moore School of the U. of Pennsylvania, ENIAC

John Von Neumann

Princeton U.
EDVAC, the blueprint of the first stored program digital computer

Maurice Wilkes

Cambridge U., EDSAC, the first operational stored-program digital computer

John Vincent Atanasoff

Iowa State University
Designed a machine in 1939-1940 to solve differential equations. Recognition came much later.
Generations

First generation: vacuum tubes
Second generation: transistors
Third generation: integrated circuits
Fourth generation: LSI and VLSI

Measuring speed

MIPS $10^6$ instructions per second
BIPS $10^9$ instructions per second
MFLOPS $10^6$ floating point ops per second
GFLOPS $10^9$ floating point ops per second
TERAFLOPS $10^{12}$ floating point ops per second
PETAFL OPS $10^{15}$ floating point ops per second
**Units of time**

1 second

1 millisecond (ms)  = 10^{-3} second

1 microsecond (μs)  = 10^{-6} second

1 nanosecond (ns)  = 10^{-9} second

1 picosecond (ps)  = 10^{-12} second

**Questions**

How much time does it take to add two integers?

How much time does your computer take to read a 1 MB (megabyte) file from a disk?

What distance does light travel in 1 nanosecond?
A Basic Digital Computer

CPU or Processor    MEMORY    I/O
Visualizing instruction execution

\[
\begin{align*}
\text{address} & \quad \text{data} \\
0 & \quad 500 & x \\
1 & \quad 24 & y \\
2 & \quad -32 & z \\
3 & \quad 0 & a
\end{align*}
\]

\[
\begin{align*}
\text{a} &= x + y - z \\
\text{load } x \text{ into } r1 \\
\text{load } y \text{ into } r2 \\
\text{load } z \text{ into } r0 \\
r3 &= r1 + r2 \\
r0 &= r3 - r0 \\
\text{store } r0 \text{ into } a
\end{align*}
\]
Assembly language instructions

Load x, r1
Load y, r2
Load z, r0
Add r3, r1, r2
Sub r0, r3, r0
Store r0, a

Each processor has a different set of registers, and different assembly language instructions. The assembly language instructions of PC and Mac are different.

Motorola 68000 has 16 registers r0-r15
MIPS has 32 registers r0-r31
Pentium has 8 general purpose & 6 segment registers.
Binary or Machine Language program

Both program and data are represented using 0’s and 1’s inside a computer. Here is a sample:

```
0 0 0 0 0 0 ...
```

Load address of x (operation code)

```
1 0 1 0 0 0 ...
```

```
0 0 0 0 0 0 ...
```

Add r3 r1 r2 unused

These are instruction formats. Each instruction has a specific format.
Representing data

How do we represent

- Signed integers
- Fractions
- Alphanumeric characters
- Floating point numbers
- Pictures?

Review

Both are bit strings.
Indistinguishable.
**Bits, bytes, words**

Bit: 0, 1

Byte: string of 8 bits. Each byte has an address.

Word: one or more bytes (usually 2 or 4 or 8).

```
0  0101000
  11110000
  00000000
  11111111
  00001111
  10111011
  00111100
  00000111
          word 0

1           word 1
  0101000
  11110000
  00000000
  11111111
  00001111
  10111011
  00111100
  00000111
```
Byte order in a word

Big endian order       [byte 0, byte 1, byte 2, byte 3]
Little endian order    [byte 3, byte 2, byte 1, byte 0]
Registers vs. memory

Data can be stored in registers or memory locations. Memory access is slower (takes approximately 50 ns) than register access (takes approximately 1 ns).

Number of registers is however quite limited. To increase the speed of computation it pays to keep the variables in registers as long as possible.
Clock

Ticks at the heart of every processor.

voltage

\[ \text{frequency } f = \frac{\text{no. of oscillations per second}}{\text{period}} = \frac{1}{\text{period}} \]

Typically most CPU operations are synchronized with this clock.

Each instruction is executed in a few clock cycles.
**MIPS registers**

**MIPS has 32 registers r0-r31. The conventional use of these registers is as follows:**

<table>
<thead>
<tr>
<th>register</th>
<th>assembly name</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>$zero</td>
<td>Always 0</td>
</tr>
<tr>
<td>r1</td>
<td>$at</td>
<td>Reserved for assembler</td>
</tr>
<tr>
<td>r2-r3</td>
<td>$v0-$v1</td>
<td>Stores results</td>
</tr>
<tr>
<td>r4-r7</td>
<td>$a0-$a3</td>
<td>Stores arguments</td>
</tr>
<tr>
<td>r8-r15</td>
<td>$t0-$t7</td>
<td>Temporaries, not saved</td>
</tr>
<tr>
<td>r16-r23</td>
<td>$s0-$s7</td>
<td>Contents saved for use later</td>
</tr>
<tr>
<td>r24-r25</td>
<td>$t8-$t9</td>
<td>More temporaries, not saved</td>
</tr>
<tr>
<td>r26-r27</td>
<td>$k0-$k1</td>
<td>Reserved by operating system</td>
</tr>
<tr>
<td>r28</td>
<td>$gp</td>
<td>Global pointer</td>
</tr>
<tr>
<td>r29</td>
<td>$sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>r30</td>
<td>$fp</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>r31</td>
<td>$ra</td>
<td>Return address</td>
</tr>
</tbody>
</table>
**Example assembly language programs**

**Example 1**

\[
f = g + h - i
\]

Assume that \( f, g, h, i \) are assigned to \( $s0, $s1, $s2, $s3 \)

\[
\begin{align*}
\text{add } &\$t0, $s1, $s2 & \# \text{ register } \$t0 \text{ contains } g + h \\
\text{sub } &\$s0, \$t0, $s3 & \# f = g + h - i
\end{align*}
\]

**Example 2.**

\[
g = h + A[8]
\]

Assume that \( g, h \) are in \( $s1, $s2 \). \( A \) is an array of words, the elements are stored in consecutive locations of the memory. The base address is stored in \( $s3 \).

\[
\begin{align*}
\text{lw } &\$t0, 32($s3) & \# t0 \text{ gets } A[8], 32 = 4 \times 8 \\
\text{add } &$s1, $s2, \$t0 & \# g = h + A[8]
\end{align*}
\]
Machine language representations

Instruction “add” belongs to the **R-type format**.

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shift amt</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

The function field is an extension of the opcode, and they together determine the operation.

Note that “sub” has a similar format.
Instruction “lw” (load word) belongs to **I-type format**.

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

\[ \begin{array}{c c c c}
\text{base} & \uparrow & \text{dst} & \uparrow \\
6 & 5 & 5 & 16 \\
\end{array} \]

lw $t0, 32($s3) will be coded as

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>19</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Both “lw” and “sw” (store word) belong to I-format.
Making decisions

\[
\text{if } (i == j) \quad f = g + h; \quad \text{else} \quad f = g - h
\]

Use \texttt{bne} = branch-nor-equal, \texttt{beq} = branch-equal, and \(j = \text{jump}\)

\[
f, g, h, \text{ are mapped into } s0, s1, s2
\]

\[
i, j \text{ are mapped into } s3, s4
\]

\[
\begin{align*}
\text{bne } s3, s4, \text{ Else} & \quad \# \text{goto Else when } i=j \\
\text{add } s0, s1, s2 & \quad \# f = g + h \\
\text{j Exit} & \quad \# \text{goto Exit} \\
\text{Else: sub } s0, s1, s2 & \quad \# f = g - h \\
\text{Exit:}
\end{align*}
\]
The program counter

Every machine has a **program counter** (called PC) that points to the next instruction to be executed.

Ordinarily, PC is incremented by 4 after each instruction is executed. A branch instruction alters the flow of control by modifying the PC.
Compiling a while loop

while (A[i] == k) i = i + j;

Initially $s3, s4, s5$ contains $i, j, k$ respectively. Let $s6$ store the base of the array $A$. Each element of $A$ is a 32-bit word.

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add $t1, s3, s3</td>
<td>$t1 = 2*i</td>
</tr>
<tr>
<td></td>
<td>add $t1, $t1, $t1</td>
<td>$t1 = 4*i</td>
</tr>
<tr>
<td></td>
<td>add $t1, $t1, s6</td>
<td>$t1 contains address of $A[i]</td>
</tr>
<tr>
<td></td>
<td>lw $t0, 0($t1)</td>
<td>$t0 contains $A[i]</td>
</tr>
<tr>
<td></td>
<td>bne $t0, s5, Exit</td>
<td>goto Exit if $A[i] \neq k</td>
</tr>
<tr>
<td></td>
<td>add $s3, s3, s4</td>
<td>$i = i + j</td>
</tr>
<tr>
<td></td>
<td>j Loop</td>
<td>goto Loop</td>
</tr>
</tbody>
</table>

Exit: <next instruction>

Note the use of pointers.
Compiling a switch statement

switch (k) {
    case 0:   f = i + j; break;
    case 1:   f = g + h; break;
    case 2:   f = g - h; break;
    case 3:   f = I - j; break;
}

Assume, $s0-$s5 contain f, g, h, i, j, k.
Assume $t2$ contains 4.

slt $t3$, $s5$, $zero  \quad$ # if k<0 then $t3 = 1$ else $t3=0$
bine $t3$, $zero$, Exit   \quad$ # if k<0 then Exit
slt $t3$, $s5$, $t2  \quad$ # if k<4 then $t3 = 1$ else $t3=0$
beq $t3$, $zero$, Exit   \quad$ # if k\geq 4 the Exit

What next? Jump to the right case!
Base address of the jumptable

register $t4

32-bit address L0
32-bit address L1
32-bit address L2
32-bit address L3

f = i + j
J Exit

f = g+h

j Exit

MEMORY

Exit
Here is the remainder of the program:

    add $t1, $s5, $s5
    add $t1, $t1, $t1
    add $t1, $t1, $t4
    lw $t0, 0($t1)
    jr $t0
L0: add $s0, $s3, $s4
     J Exit
L1: add $s0, $s1, $s2
     J Exit
L2: sub $s0, $s1, $s2
     J Exit
L3: sub $s0, $s3, $s4
     Exit: <next instruction>
The instruction format for jump

\[ J\ 10000 \] is represented as

\[
\begin{array}{cc}
2 & 2500 \\
6\text{-bits} & 26\text{ bits}
\end{array}
\]

This is the J-type format of MIPS instructions. The most significant 4-bits of the PC remain unchanged.

Conditional branch is represented using I-type format:

\[ \text{bne}\ $s0,\ $s1,\ 1234 \] is represented as

\[
\begin{array}{ccc}
5 & 16 & 17 \\
6 & 5 & 5
\end{array}
\]

16-bit offset

\[ \text{PC} + 4 + \text{offset} \text{ determines the branch target.} \]

This is called \textbf{PC-relative addressing.}
Revisiting machine language of MIPS

Loop:  
add $t1, $s3, $s3  # starts from 80000
add $t1, $t1, $t1
add $t1, $t1, $s6
lw  $t0, 0($t1)
bne $t0, $s5, Exit
add $s3, $s3, $s4
j  Loop

Exit:

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>80000</td>
<td>0</td>
<td>19</td>
<td>19</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>80004</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>80008</td>
<td>0</td>
<td>9</td>
<td>22</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>80012</td>
<td>35</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>80016</td>
<td>5</td>
<td>8</td>
<td>21</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>80020</td>
<td>0</td>
<td>19</td>
<td>20</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>80024</td>
<td>2</td>
<td></td>
<td>20000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80028</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R-type  I-type  J-type
Addressing Modes

What are the different ways to access an operand?

- **Register addressing**
  
  Operand is in register
  
  \[ \text{add } \$s1, \$s2, \$s3 \text{ means } \$s1 \leftarrow \$s2 + \$s3 \]

- **Base addressing**
  
  Operand is in memory.
  
  The address is the sum of a register and a constant.
  
  \[ \text{lw } \$s1, 32(\$s3) \text{ means } \$s1 \leftarrow M[\$s3 + 32] \]

  As special cases, you can implement

  - **Direct addressing**
    
    \[ \$s1 \leftarrow M[32] \]

  - **Indirect addressing**
    
    \[ \$s1 \leftarrow M[\$s3] \]

    Which helps implement pointers.
• **Immediate addressing**
  
The operand is a constant.
  
  *How can you execute* $s1 \leftarrow 7$?
  
  addi $s1$, $zero$, 7 means $s1 \leftarrow 0 + 7$
  
  (add immediate, uses the I-type format)

• **PC-relative addressing**
  
The operand address = PC + an offset
  
  Implements position-independent codes. A small offset is adequate for short loops.
Pseudoinstructions

These are simple assembly language instructions that do not have a direct machine language equivalent. During assembly, each pseudoinstruction is translated by the assembler into one or more machine language instructions.

Example

move $t0, $t1  # $t0 ← $t1

Implemented as  add $t0, $zer0, $t1

blt $s0, $s1, label  # if $s0 < $s1 then goto label

Implemented as

slt $t0, $s0, $s1  # if $s0 < $s1 then $t0 = 1 else $t0 = 0
bne $t0, $zero, label  # if $t0 ≠ 0 then goto label

Pseudoinstructions give MIPS a richer set of assembly language instructions.
Assembler Directives

An assembler expects an assembly language program to be presented in a certain format for proper interpretation.

```
.data    #Begins the data segment
age     .byte  25
align   2     {align next line to a 2^2 word boundary}
salary  .word  50000
name    .asciiz peter brookes

.text    #Begins the code segment
.align 2
.globl main
main:    lw $t0, 40 ($s3)
{rest of the program}
```
Procedure Call

Main

procedure

Uses a stack. What is a stack?
The stack

Occupies a part of the main memory. In MIPS, it grows from high address to low address as you push data on the stack. Consequently, the content of the stack pointer ($sp) decreases.
Use of the stack in procedure call

Before the subroutine executes, save registers.

Jump to the subroutine using jump-and-link (jal address)

(jal address means ra $ PC+4; PC $ address)

After the subroutine executes, restore the registers.

Return from the subroutine using jr (jump register)

(jr ra means PC $ Memory content from the address stored in register ra)

Example

int leaf (int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}

The arguments g, h, i, j are put in $a0-$a3.
The result f is put into $s0, and returned to $v0.
The structure of the procedure

The calling program calls procedure leaf by \texttt{jal leaf}

Leaf:

\begin{verbatim}
subi $sp, $sp, 12  # $sp = $sp-12, make room
sw $t1, 8($sp)    # save $t1 on stack
sw $t0, 4($sp)    # save $t0 on stack
sw $s0, 0($sp)    # save $s0 on stack
\end{verbatim}

Now we can use the registers $t1, $t0, $s0 in the body of the procedure.

\begin{verbatim}
add $t0, $a1, $a2   # $t0 = g + h
add $t1, $a2, $a3   # $t1 = i + j
sub $s0, $t0, $t1   # $s0 = (g + h) – (i + j)
\end{verbatim}

Return the result into the register $v0.

\begin{verbatim}
add $v0, $s0, $zero # returns $f = (g+h)-(i+j)$ to $v0
\end{verbatim}
Now restore the old values of the registers by popping the stack.

\[
\begin{align*}
\text{lw } & \quad \text{s0, 0(sp)} \quad \# \text{ restore s0} \\
\text{lw } & \quad \text{t0, 4(sp)} \quad \# \text{ restore t0} \\
\text{lw } & \quad \text{t1, 8(sp)} \quad \# \text{ restore t1} \\
\text{addi } & \quad \text{sp, sp, 12} \quad \# \text{ adjust sp}
\end{align*}
\]

Finally, return to the main program.

\[
\text{jr } \quad \text{ra} \quad \# \text{ return to caller.}
\]

By convention, the $t$ registers need not be saved. This simplifies the codes.
A recursive procedure

Example. Compute factorial (n)

```c
int fact (int n)
{
    if (n < 1) return (1);
    else return (n * fact(n-1))
}
```

(Plan) Put n in $a0. Result should be available in $v0.
calling program

procedure fact

..., ...  
\textit{a0} = \textit{n} (3)  
\textit{j}al \textit{fact}(4000)  
\textit{read} \textit{fact(n)} from \textit{v0}  

\begin{align*}
\text{push ra} \\
\text{push a0} \\
\text{if } \textit{n} < 1 \text{ then } \{ \textit{v0} = 1 \text{ } \\
\text{Return to ra} \} \\
\textit{a0} = \textit{n} - 1 \\
\text{jal fact}(4000) \\
\textit{v0} = \text{old } \textit{a0} \times \text{fact}(\textit{n} - 1) \\
\text{return to old ra}
\end{align*}

\begin{tabular}{|c|}
\hline
\textit{ra} = 1004 \\
\textit{a0} = 3 \\
\textit{ra} = 4024 \\
\textit{a0} = 2 \\
\textit{ra} = 4024 \\
\textit{a0} = 1 \\
\hline
\end{tabular}

\begin{tabular}{|c|}
\hline
\textit{a0} & \textbf{3} \\
\hline
\textit{v0} & \textbf{result} \\
\hline
\end{tabular}
Now test if \( n < 1 \) (i.e. \( n = 0 \)). In that case return 0 to \( v0 \).

```
slti $t0, $a0, 1  # if \( n \geq 1 \) then goto L1
beq $t0, $zero, L1
addi $v0, $zero, 1  # return 1 to \( v0 \)
addi $sp, $sp, 8    # pop 2 items from stack
jr   $ra           # return
```

L1:
```
subi $a0, $a0, 1    # decrement \( n \)
jal fact           # call fact with (\( n - 1 \))
```

Now, we need to compute \( n \times \text{fact}(n-1) \)

```
lw $a0, 0($sp)      # restore argument \( n \)
lw $ra, 4($sp)     # restore return address
addi $sp, $sp, 8   # pop 2 items
mult $v0, $a0, $v0 # return \( n \times \text{fact}(n-1) \)
jr   $ra           # return to caller
```