9 September 2004

- Questions?
- Programming Continued

So far we’ve learned:

- MIPS — loading words but addressing bytes
  — arithmetic on registers only

**Instruction**

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a1 = $a2 + $a3</td>
<td>add $a1, $a2, $a3</td>
</tr>
<tr>
<td>$a1 = $a2 - $a3</td>
<td>sub $a1, $a2, $a3</td>
</tr>
<tr>
<td>$a1 = Memory[$a2+100]</td>
<td>lw $a1, 100($a2)</td>
</tr>
<tr>
<td>Memory[$a2+100] = $a1</td>
<td>sw $a1, 100($a2)</td>
</tr>
</tbody>
</table>

---

### Execution Example

**Program Counter**

<table>
<thead>
<tr>
<th>200</th>
<th>Memory (32 bits)</th>
<th>Memory (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>112</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Registers (32 bits)**

<table>
<thead>
<tr>
<th>6</th>
<th>1234</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>120</td>
</tr>
<tr>
<td>9</td>
<td>-314159</td>
</tr>
<tr>
<td>10</td>
<td>316</td>
</tr>
</tbody>
</table>

**Instruction Register (32 bits)**

<table>
<thead>
<tr>
<th>op</th>
<th>rd</th>
<th>rt</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

**Instruction Register**

<table>
<thead>
<tr>
<th>10011 01000 01000 00000000000000</th>
</tr>
</thead>
</table>

---

### Execution Example: Fetch (200)

**Program Counter**

<table>
<thead>
<tr>
<th>200</th>
<th>Memory (32 bits)</th>
<th>Memory (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>112</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Registers**

<table>
<thead>
<tr>
<th>6</th>
<th>1234</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>120</td>
</tr>
<tr>
<td>9</td>
<td>-314159</td>
</tr>
<tr>
<td>10</td>
<td>316</td>
</tr>
</tbody>
</table>

**Instruction Register**

<table>
<thead>
<tr>
<th>10011 01000 01000 00000000000000</th>
</tr>
</thead>
</table>

---

### Execution Example: Execute (200)

**Program Counter**

<table>
<thead>
<tr>
<th>200</th>
<th>Memory (32 bits)</th>
<th>Memory (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>112</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Registers**

<table>
<thead>
<tr>
<th>6</th>
<th>1234</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>120</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>316</td>
</tr>
</tbody>
</table>

**Instruction Register**

<table>
<thead>
<tr>
<th>10011 01000 01000 00000000000000</th>
</tr>
</thead>
</table>

---

### Execution Example: Fetch (204)

**Program Counter**

<table>
<thead>
<tr>
<th>200</th>
<th>Memory (32 bits)</th>
<th>Memory (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>112</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Registers**

<table>
<thead>
<tr>
<th>6</th>
<th>1234</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>120</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>316</td>
</tr>
</tbody>
</table>

**Instruction Register**

<table>
<thead>
<tr>
<th>000000 01000 01111 01101 100000 100000</th>
</tr>
</thead>
</table>

---

### Execution Example: Execute (204)

**Program Counter**

<table>
<thead>
<tr>
<th>200</th>
<th>Memory (32 bits)</th>
<th>Memory (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>112</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Registers**

<table>
<thead>
<tr>
<th>6</th>
<th>1234</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>120</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>316</td>
</tr>
</tbody>
</table>

**Instruction Register**

<table>
<thead>
<tr>
<th>000000 01000 01111 01101 100000 100000</th>
</tr>
</thead>
</table>

---

• 1
Execution Example: Execute(204)

Program Counter

| 200 | 204 | 208 |

Memory

| 112 | 123 | 124 |

Instructions

- lw $9, 0($8)
- add $9, $9, $7
- sw $9, 8($8)

Registers

| 6 | 7 | 8 | 9 | 10 |

| 0 | 23 | 120 | 0 | 316 |

Instruction Register

Control

- Decision making instructions
  - alter the control flow,
  - change the “next” instruction to be executed by changing the PC
- MIPS conditional branch instructions:
  - bne $t0, $t1, Label
  - beq $t0, $t1, Label

Example:

```
if (i==j) h = i + j;
```

MIPS unconditional branch instructions:

```
jal label
jr $ra
```

Example:

```
if (i>3)
  beq $s4, $s5, Label1
else
  j Label2
```

Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

```
slt $t0, $s1, $s2
```

- Can use this instruction to build "bit $s1, $s2, Label"
- Note that the assembler needs a register to do this, there are policy of use conventions for registers
So far:

- **Instruction**
  - `add $s1,$s2,$s3` $s1 = $s2 + $s3
  - `lw $s1,100($s2)` $s1 = Memory[$s2+100]
  - `bne $s4,$s5,$j` Next instr is at Label if $s4 != $s5
  - `j $j` Label Next instr at label, save return addr
  - `jr $r` Next instr is addr in $r

- **Formats:**
  - B op rs rt rd shamt funct
  - I op rs rt 16-bit address
  - J op rs rt 26-bit address

---

Policy of Use Conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r0</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$r1</td>
<td>2</td>
<td>return for result and expression evaluator</td>
</tr>
<tr>
<td>$r2-5</td>
<td>4,5</td>
<td>arguments</td>
</tr>
<tr>
<td>$r6-11</td>
<td>6,7</td>
<td>arguments</td>
</tr>
<tr>
<td>$r12</td>
<td>12</td>
<td>saved</td>
</tr>
<tr>
<td>$a0-7</td>
<td>16-15</td>
<td>more temporary</td>
</tr>
<tr>
<td>$t0-7</td>
<td>24-25</td>
<td>local pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>28</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>

---

Stack Pointer?

- Register $sp is used to keep track of the stack
- Nothing special about $sp in the hardware, just a convention
- Stack starts at the TOP of memory and grows DOWN (Why?)
- Allocate space on the stack by decrementing $sp
- Free space on the stack by incrementing $sp
- Used to save return address and temporary variables as well as for local variables and arrays

---

Memory Layout

- Stack Pointer
- Reserve
- Static Data
- Text
- Reserved
- Dynamically loaded

---

Making code work!

- This example is available online.
- You should write algorithms in C/JAVA first
  - Then HAND translate...

---

C to Assembler

```c
char S1[] = "This is a string."
char F0(16);
void main()
{
    strcpy(F0, S1);
}
void strcpy(char* dst, char* src)
{
    int i = 0;
    while((dst[i] = src[i]) != 0)
        i = i+1;
}
```
#char S1[] = "This is a string."
#define Foo[16]
#define S2[100]

.data
S1: .ascii "This is a string."
Foo: .space 16 # some junk
S2: .space 100 # destination string

.text
.globl main
main:
add $sp, $sp, -4 # get space on the stack
sw $ra, 0($sp) # save main's return address
lb $a0, S2 # address of S2 in the first argument
lb $a1, S1 # address of S1 in the second argument
jal strcpy # call strcpy
lw $ra, 0($sp) # restore main's return address
add $sp, $sp, 4 # restore the stack pointer
jr $ra # main

void strcpy(char* dst, char* src) {
    # i = 0;
    # while((dst[i] = src[i]) != 0)
    #      i = i+1;
}

strcpy:
    move $t0, $zero # i in $t0 = 0
    li $t1, $t0, $t0 # address of src[i] in $t1
    li $t2, 0($t1) # t2 = src[i]
    add $t1, $t0, $t2 # address of dst[i] in $t1
    sb $t0, 0($t1) # dst[i] = src[i]
    add $t0, $t0, 1 # i = i+1
    beq $t2, $zero, L3 if dst[i] != 0 repeat
    jr $ra # return

C O D E  P a t t e r n  f o r  I F

if(COND_EXPR) { STMTS1 } else { STMTS2 }

CONDCODE(COND_EXPR, Lfalse1, Ltrue2)
Ltrue2:
    CODE(STMTS1)
Lfalse1:
    CODE(STMTS2)
Lnext3:

C O D E  P a t t e r n  f o r  C o n d i t i o n a l  E x p r

Comparisons: = = ! = < > < = > =
Conjunctions: && ||
Parenthesis: ()

A = = B
get left in reg A
get right in reg B
bx $a, $b, Lfalse
Lfalse:
A < B
get left in reg A
get right in reg B
slt $t0, $a, $b
beq $t0, $zero, Lfalse
Ltrue2:
**CP for && and ||**

L & & R

L | R

---

**Example Conditional**

\[(A \&\& B \&\& A + C) \{ ST \} else \{ SE \}\]

---

**While**

\[while(\text{COND\_EXPR}) \{ \text{STMTS} \};\]

\[\text{while44:}\]

CONDCODE(\text{COND\_EXPR}, Lfalse45, Ltrue46)

\[\text{Ltrue46:}\]

CODE(\text{STMTS})

\[\text{Lfalse45:}\]

---

**FOR**

\[\text{for}(\text{INIT}; \text{COND\_EXPR}; \text{UPDATE}) \{ \text{STMTS} \}\]

\[\text{for17:}\]

CONDCODE(\text{COND\_EXPR}, Lnext18, Ltrue19)

\[\text{Ltrue19:}\]

CODE(\text{STMTS})

CODE(\text{UPDATE})

\[\text{Lnext18:}\]

---

**Functions**

\[\text{int foo}(\text{int} a, \text{int} b, \text{int} c, \text{int} d) \{ \text{STMTS}; \text{return EXPR;} \}\]

we know a=&$a0, b=&$a1, c=&$a2, d=&$a3
assign other simple variables to registers as possible

foo:

save any of $sra or $s0-s7 that are overwritten

CODE(\text{STMTS})

leave result in $v0

restore $sra, and $s0-s7 if we saved them earlier

jr $sra

---

**cfind**

\[\text{int cfind}(\text{char str[]}, \text{char} c) \{\]

for(int i=0; str[i] != 0; i++)

if(str[i] == c) return i;

return -1;

\}

a0 == address of str

a1 == value of character c

v0 == i

no need to save anything
Expand return

cfind:  
move $v0, $zero  
Lfor1:  
   add $t0, $a0, $v0  # address of s[i]  
   lb $t1, 0($t0)  # $t1 = s[i]  
   beq $t1, $zero, Lnext2  
Ltrue3:  
   bne $t1, $a1, Lfalse4  
Ltrue5:  
   jr $ra  # return i already in v0  
   j Lfor1  
Lnext2:  
   cfunc('i+1')  
   jr $ra  

Expand i++

cfind:  
move $v0, $zero  
Lfor1:  
   add $t0, $a0, $v0  # address of s[i]  
   lb $t1, 0($t0)  # $t1 = s[i]  
   beq $t1, $zero, Lnext2  
Ltrue3:  
   bne $t1, $a1, Lfalse4  
Ltrue5:  
   jr $ra  # return i already in v0  
   j Lfor1  
Lnext2:  
   cfunc('return -1')  
   jr $ra  

Expand return -1

cfind:  
move $v0, $zero  
Lfor1:  
   add $t0, $a0, $v0  # address of s[i]  
   lb $t1, 0($t0)  # $t1 = s[i]  
   beq $t1, $zero, Lnext2  
   bne $t1, $a1, Lfalse4  
Ltrue5:  
   jr $ra  # return i already in v0  
   addi $v0, $v0, 1  # i = i+1;  
   jr $ra  
Lfalse4:  
   subi $v0, $zero, 1  # return value = -1  
   jr $ra  

Remove unused labels

cfind:  
move $v0, $zero  # i = 0  
Lfor1:  
   add $t0, $a0, $v0  # address of s[i]  
   lb $t1, 0($t0)  # $t1 = s[i]  
   beq $t1, $zero, Lnext2  # if s[i] == 0  
   bne $t1, $a1, Lfalse4  # if s[i] == c  
   jr $ra  # return i already in v0  
Lfalse4:  
   addi $v0, $v0, 1  # i = i+1;  
   jr $for1  
Lnext2:  
   sub $v0, $zero, 1  # return -1  
   jr $ra