Today’s Lecture

Admin
• HW #1 is due
• HW #2 assigned

Outline
• Review
• A specific ISA, we'll use it throughout semester, very similar to the NiosII ISA (we will use for programs)
• Instruction categories
• Specific Instructions

Reading
Chapter 2, Appendix A (on CD), “The Nios Soft Processor” Sections 3, 5-8
Review: Basic ISA Classes

**Accumulator:**
- 1 address: add A, acc ← acc + mem[A]
- 1+x address: addx A, acc ← acc + mem[A + x]

**Stack:**
- 0 address: add, tos ← tos + next (JAVA VM)

**General Purpose Register:**
- 2 address: add A B, A ← A + B
- 3 address: add A B C, A ← B + C

**Load/Store:**
- 3 address: add Ra Rb Rc, Ra ← Rb + Rc
  - load Ra Rb, Ra ← mem[Rb]
  - store Ra Rb, mem[Rb] ← Ra

Review: LOAD / STORE ISA

- **Instruction set:**
  - add, sub, mult, div, … only on operands in registers
  - ld, st, to move data from and to memory, only way to access memory

**Example:** \( a \times b - (a + c \times b) \) (assume in memory)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r1, c</td>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>ld r2, b</td>
<td>2, 3</td>
<td>b</td>
</tr>
<tr>
<td>mult r1, r2</td>
<td>6, 3</td>
<td>c</td>
</tr>
<tr>
<td>ld r3, a</td>
<td>6, 3, 4</td>
<td></td>
</tr>
<tr>
<td>add r1, r1, r3</td>
<td>10, 3, 4</td>
<td></td>
</tr>
<tr>
<td>mult r2, r2, r3</td>
<td>10, 12, 4</td>
<td></td>
</tr>
<tr>
<td>sub r3, r2, r1</td>
<td>10, 12, 2</td>
<td></td>
</tr>
</tbody>
</table>

7 instructions
Using Registers to Access Memory

• Registers can hold memory addresses

Given

```c
int x; int *p;
p = &x;
*p = *p + 8;
```

Instructions

```c
ld r1, p // r1 <- mem[p]
ld r2, r1  // r2 <- mem[r1]
add r2, r2, 0x8 // increment x by 8
st r1, r2 // mem[r1] <- r2
```

• Many different ways to address operands

➢ not all Instruction sets include all modes

Making Instructions Machine Readable

• So far, still too abstract
  ➢ add r1, r2, r3

• Need to specify instructions in machine readable form

• Bunch of Bits

• Instructions are bits with well defined fields
  ➢ Like a floating point number has different fields

• Instruction Format
  ➢ establishes a mapping from “instruction” to binary values
  ➢ which bit positions correspond to which parts of the instruction (operation, operands, etc.)
### Example: MIPS

#### Register-Register

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ox</td>
</tr>
</tbody>
</table>

#### Register-Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
</tbody>
</table>

#### Branch

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Opx</td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
</tbody>
</table>

#### Jump / Call

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>target</td>
</tr>
</tbody>
</table>

---

### Stored Program Computer

- **Instructions**: a fixed set of built-in operations
- **Instructions and data are stored in the (same) computer memory**
- **Fetch-Execute Cycle**
  
  ```
  while (!done)
  {
  fetch instruction
  execute instruction
  }
  ```
- **This is done by the hardware for speed**
- **This is what the NiosII Instruction Set Simulator does**
What Must be Specified?

- **Instruction Format**
  - how do we tell what operation to perform?

- **Location of operands and result**
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?

- **Data type and Size**

- **Operations**
  - what are supported

- **Successor instruction**
  - jumps, conditions, branches

- *fetch-decode-execute is implicit!*

---

**MIPS ISA Categories**

- **Arithmetic**
  - add, sub, mul, etc

- **Logical**
  - and, or, shift

- **Data Transfer**
  - load, store
  - MIPS is LOAD/STORE architecture

- **Conditional Branch**
  - implement if, for, while... statements

- **Unconditional Jump**
  - support method invocation (function call, procedure calls)
MIPS Instruction set Architecture

- 3-Address Load/Store Architecture.
- Register and Immediate addressing modes for operations.
- Immediate and Displacement addressing for Loads and Stores.

Examples (Assembly Language):

- `add $1, $2, $3` # $1 = $2 + $3
- `addi $1, $1, 4` # $1 = $1 + 4
- `lw $1, 100 ($2)` # $1 = Memory[$2 + 100]
- `sw $1, 100 ($2)` # Memory[$2 + 100] = $1
- `lui $1, 100` # $1 = 100 \times 2^{16}
- `addi $1, $3, 100` # $1 = $3 + 100

MIPS Integer Registers

- Registers: fast memory, integral part of the CPU.
- Programmable storage: 2^{32} bytes
- 31 x 32-bit GPRs (R0 = 0)
- 32 x 32-bit FP regs (paired DP)
- 32-bit HI, LO, PC
### MIPS Instruction Formats

**R-type: Register-Register**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>22</th>
<th>21</th>
<th>17</th>
<th>16</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs</td>
<td>Rt</td>
<td>Rd</td>
<td>shamt</td>
<td>func</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**I-type: Register-Immediate**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>22</th>
<th>21</th>
<th>17</th>
<th>16</th>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs</td>
<td>Rt</td>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**J-type: Jump / Call**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td></td>
<td>target</td>
<td></td>
</tr>
</tbody>
</table>

**Terminology**

- Op = opcode
- Rs, Rt, Rd = register specifier

### NiosII Instruction Formats

**R-type: Register-Register**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>22</th>
<th>21</th>
<th>17</th>
<th>16</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>OPX</td>
<td>Op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**I-type: Register-Immediate**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>22</th>
<th>21</th>
<th>6</th>
<th>5</th>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td>IMMEDI6</td>
<td>Op</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**J-type: Jump / Call**

<table>
<thead>
<tr>
<th>31</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMMEDI26</td>
<td>Op</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Terminology**

- Op = opcode
- Rs, Rt, Rd = register specifier
Operand Addressing: Register direct

Example: ADD $1, $2, $3  # $1 = $2 + $3

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shmt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>00010</td>
<td>00011</td>
<td>00001</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

Immediate: 16 bit value

Operand Addressing:
Register Direct and Immediate

Add Immediate Example
addi  $1, $2, 100  # $1 = $2 + 100

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>00010</td>
<td>00001</td>
<td>0000 0000 0110 0100</td>
</tr>
</tbody>
</table>
I-Type \langle op \rangle \ rt, rs, \ immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs</td>
<td>Rt</td>
<td>Immediate</td>
<td></td>
</tr>
</tbody>
</table>

Register +

Memory

Load Word Example

\texttt{lw \$1, 100(\$2)} \quad \# \$1 = \text{Mem}\[\$2+100]\]

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>010011</td>
<td>00010</td>
<td>00001</td>
<td>0000 0000 0110 0100</td>
</tr>
</tbody>
</table>

Successor Instruction

\begin{verbatim}
main()
{
  int x,y,same; \quad // \$0 == 0 always
  x = 43; \quad // \text{addi} \$1, \$0, 43
  y = 2; \quad // \text{addi} \$2, \$0, 2
  same = 0; \quad // \text{addi} \$3, \$0, 0
  if (x == y)
  {
    same = 1; \quad // execute only if \( x == y \)
    same = 1; \quad // addi \$3, \$0, 1
  }
}
\end{verbatim}
The Program Counter (PC)

- Special register (PC) that points to instructions
- Contains memory address (like a pointer)
- Instruction fetch is
  - \[ \text{inst} = \text{mem}[\text{pc}] \]
- To fetch next sequential instruction PC = PC + ?
  - Size of instruction?

---

x = 43;  // addi $1, $0, 43
y = 2;   // addi $2, $0, 2
same = 0; // addi $3, $0, 0
if (x == y)
  same = 1;  // addi $3, $0, 1 execute if x == y

PC is always automatically incremented to next instruction

Clearly, this is not correct
We cannot always execute both 0x10008 and 0x1000c
PC relative addressing

Branch Not Equal Example

```
bne $1, $2, 100  # If ($1!= $2) goto [PC+4+100]
```

• +4 because by default we increment for sequential
  ➢ more detailed discussion later in semester

```
op   rs   rt  immediate
000101 00001 00010 0000 0000 0110 0100
```

The Program Counter

```
x = 43;  // addi $1, $0, 43
y = 2;   // addi $2, $0, 2
same = 0; // addi $3, $0, 0
if (x == y)
  same = 1; // addi $3, $0, $1 execute if x == y
x = x + y; // addi $1, $1, $2
```

Understand branches
Successor Instruction

```c
int equal(int a1, int a2) {
    int tsame;
    tsame = 0;
    if (a1 == a2)  // only if a1 == a2
        tsame = 1;
    return(tsame);
}
```

```c
main() {
    int x, y, same;  // r0 == 0 always
    x = 43;  // addi $1, $0, 43
    y = 2;  // addi $2, $0, 2
    same = equal(x, y);  // need to call function
    // other computation
}
```

The Program Counter

- Branches are limited to 16 bit immediate
- Big programs?

```
x = 43;  // addi $1, $0, 43
y = 2;  // addi $2, $0, 2
same = equal(x, y);
```

```
0x10000  addi $1, $0, 43
0x10004  addi $2, $0, 2
0x10008  "go execute equal"
```

```
0x30408  addi $3, $0, 0
0x3040c  beq $1, $2, 8
0x30410  addi $3, $0, 1
          "return $3"
```
Jump and Link Example

**JAL 1000**  # PC<- 1000, $31<-PC+4

$31 set as side effect, used for returning, implicit operand

---

Jump Register Example

**jr $31**  # PC <- $31
Instructions for Procedure Call and Return

```c
int equal(int a1, int a2) {
    int tsame;
    tsame = 0;
    if (a1 == a2)
        tsame = 1;
    return(tsame);
}
main()
{
   int x,y,same;
   x = 43;
   y = 2;
   same = equal(x,y);
   // other computation
}
```

MIPS Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant</td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands</td>
</tr>
<tr>
<td>add imm. unsign.</td>
<td>addiu $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant</td>
</tr>
<tr>
<td>multiply</td>
<td>mul $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>mul $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td>divide</td>
<td>div $2,$3</td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3</td>
<td>Unsigned quotient</td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3</td>
<td>Unsigned remainder</td>
</tr>
<tr>
<td>Move from Hi</td>
<td>mfhi $1</td>
<td>$1 = Hi</td>
<td>Used to get copy of Hi</td>
</tr>
<tr>
<td>Move from Lo</td>
<td>mflo $1</td>
<td>$1 = Lo</td>
<td>Used to get copy of Lo</td>
</tr>
</tbody>
</table>

Which add for address arithmetic? Which for integers?
**MIPS Logical Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and $1,$2,$3</td>
<td>$1 = $2 &amp; $3</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>or</td>
<td>or $1,$2,$3</td>
<td>$1 = $2</td>
<td>$3</td>
</tr>
<tr>
<td>xor</td>
<td>xor $1,$2,$3</td>
<td>$1 = $2 \oplus $3</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td>nor</td>
<td>nor $1,$2,$3</td>
<td>$1 = \neg(2 \oplus 3)$</td>
<td>Bitwise NOR</td>
</tr>
<tr>
<td>and immediate</td>
<td>andi $1,$2,10</td>
<td>$1 = 2 &amp; 10</td>
<td>Bitwise AND reg, const</td>
</tr>
<tr>
<td>or immediate</td>
<td>ori $1,$2,10</td>
<td>$1 = 2</td>
<td>10</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xori $1,$2,10</td>
<td>$1 = \neg 2 &amp; \neg 10</td>
<td>Bitwise XOR reg, const</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sil $1,$2,10</td>
<td>$1 = 2 \ll 10</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srl $1,$2,10</td>
<td>$1 = 2 \gg 10</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift right arith.</td>
<td>sra $1,$2,10</td>
<td>$1 = 2 \gg 10</td>
<td>Shift right (sign extend)</td>
</tr>
<tr>
<td>shift left logical</td>
<td>silv $1,$2,$3</td>
<td>$1 = 2 \ll 3</td>
<td>Shift left by var</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srlv $1,$2,$3</td>
<td>$1 = 2 \gg 3</td>
<td>Shift right by var</td>
</tr>
<tr>
<td>shift right arith.</td>
<td>srav $1,$2,$3</td>
<td>$1 = 2 \gg 3</td>
<td>Shift right arith. by var</td>
</tr>
</tbody>
</table>

**MIPS Data Transfer Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW R3, 500(R4)</td>
<td>Store word</td>
</tr>
<tr>
<td>SH R3, 502(R2)</td>
<td>Store half</td>
</tr>
<tr>
<td>SB R2, 41(R3)</td>
<td>Store byte</td>
</tr>
<tr>
<td>LW R1, 30(R2)</td>
<td>Load word</td>
</tr>
<tr>
<td>LH R1, 40(R3)</td>
<td>Load halfword</td>
</tr>
<tr>
<td>LHU R1, 40(R3)</td>
<td>Load halfword unsigned</td>
</tr>
<tr>
<td>LB R1, 40(R3)</td>
<td>Load byte</td>
</tr>
<tr>
<td>LBU R1, 40(R3)</td>
<td>Load byte unsigned</td>
</tr>
<tr>
<td>LUI R1, 40</td>
<td>Load Upper Immediate (16 bits shifted left by 16)</td>
</tr>
</tbody>
</table>

Why do we need LUI?

![LUI R5](image)
MIPS Compare and Branch

Compare and Branch

- **beq** rs, rt, offset  
  if $R[rs] == R[rt]$ then PC-relative branch
- **bne** rs, rt, offset  
  $R[rs] != R[rt]$

Compare to zero and Branch

- **blez** rs, offset  
  if $R[rs] <= 0$ then PC-relative branch
- **bgtz** rs, offset  
  if $R[rs] > 0$ then branch
- **bltz** rs, offset  
  if $R[rs] < 0$ then branch
- **bgez** rs, offset  
  if $R[rs] >= 0$ then branch
- **bltzal** rs, offset  
  if $R[rs] < 0$ then branch and link (into $R31$)
- **bgezl** rs, offset  
  if $R[rs] >= 0$ then branch and link (into $R31$)

- Remaining set of compare and branch take two instructions
- Almost all comparisons are against zero!

MIPS jump, branch, compare instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch on equal</td>
<td>beq $1$, $2$, 100</td>
<td>if ($1 == $2$ go to PC+4+100</td>
</tr>
<tr>
<td>Equal test; PC relative branch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>branch on not eq.</td>
<td>bne $1$, $2$, 100</td>
<td>if ($1!= $2$ go to PC+4+100</td>
</tr>
<tr>
<td>Not equal test; PC relative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>set on less than</td>
<td>slt $1$, $2$, $3$</td>
<td>if ($2 &lt; $3$ $1=1$; else $1=0</td>
</tr>
<tr>
<td>Compare less than; 2's comp.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>set less than imm.</td>
<td>sli $1$, $2$, 100</td>
<td>if ($2 &lt; 100$ $1=1$; else $1=0</td>
</tr>
<tr>
<td>Compare &lt; constant; 2's comp.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>set less than uns.</td>
<td>siltu $1$, $2$, $3$</td>
<td>if ($2 &lt; $3$ $1=1$; else $1=0</td>
</tr>
<tr>
<td>Compare &lt; constant; natural numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>set l. t. imm. uns.</td>
<td>siltui $1$, $2$, 100</td>
<td>if ($2 &lt; 100$ $1=1$; else $1=0</td>
</tr>
<tr>
<td>Compare &lt; constant; natural numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jump</td>
<td>j 10000</td>
<td>go to 10000</td>
</tr>
<tr>
<td>Jump to target address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jump register</td>
<td>jr $31$</td>
<td>go to $31</td>
</tr>
<tr>
<td>For switch, procedure return</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jump and link</td>
<td>jal 10000</td>
<td>$31 = PC + 4; go to 10000</td>
</tr>
<tr>
<td>For procedure call</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Signed v.s. Unsigned Comparison

R1= 0...00 0000 0000 0000 0001
R2= 0...00 0000 0000 0000 0010
R3= 1...11 1111 1111 1111 1111

• After executing these instructions:
  slt  r4,r2,r1
  slt  r5,r3,r1
  sltu r6,r2,r1
  sltu r7,r3,r1

• What are values of registers r4 - r7? Why?
  r4 =  ; r5 =  ; r6 =  ; r7 =  

Summary

• MIPS has 5 categories of instructions
  ➢ Arithmetic, Logical, Data Transfer, Conditional Branch, Unconditional Jump

• 3 Instruction Formats

• NiosII Soft Processor and ISA Reference

Next Time
• Assembly Programming

Reading
• Ch. 2, Appendix A, NiosII Soft Processor

• HW #2