Instruction Set Architectures 1

Introduction to Computer Architecture
David Black-Schaffer

Contents

- Review of how a program executes
- Types of instructions in MIPS
- Memory access in MIPS
  - Data operations in detail
    - Add, sub, etc.
  - Data transfers in detail
    - Load, store
  - Sequencing in detail
    - Jump, branch if equal

Review: Programming a processor

Today we're going to learn the details!
Review: walking through the program

- What will the processor do?
  1. Load the instruction
  2. Figure out what operation to do
  3. Figure out what data to use
  4. Do the computation
  5. Figure out next instruction
- Repeat this over and over and over...

In this lecture you will see many of the real details for the MIPS processor.

Example MIPS instructions

- General 3-operand format:
  - op dest, src1, src2
- Addition
  - add a, b, c
  - add a, b, 12
- Subtraction
  - sub a, b, c
- Complex: f = (g + h) - (i + j)
  = add t0, g, h
  = add t1, i, j
  = sub f, t0, t1

In this lecture you will see many of the real details for the MIPS processor.

Types of instructions

- Data operations
  - Arithmetic (add, multiply, subtract, divide)
  - Logical (and, or, not, xor, ...)
- Data transfer
  - Move (register → register)
  - Load (memory → register)
  - Store (register → memory)
- Sequencing
  - Branch (conditional, e.g., <, >, ==)
    - Jump (unconditional, e.g., goto)

Table is printed in your book for reference.
Registers in MIPS

- **32 General Purpose Registers**
  - R0: R31 or $0...$31
  - Values for instructions must come from registers

- **Some are special**
  - R0 is always zero
  - R29 is the stack pointer
  - R31 is used for procedure return addresses

- **A few special registers**
  - PC (Program Counter): current instruction
  - Hi & Lo results of multiplication
  - Floating point registers
  - A few control registers (for errors and status)

Quesion: Why is R0 always zero?
Answer: You always need zero in a program.

Memory organization

- Memory is a large 1-dimensional array
- Each location is one byte (8 bits)
- A memory address indexes into the array
- For a 32-bit computer, there are $2^{32}$ memory locations (4GB)
- For a 64-bit computer, there are $2^{64}$ memory locations (16EB)
  - 64-bit AB machines tend to be limited to ~48-bits of address space, or 4PB.

Memory words

- Most data in MIPS is handled in “words” not “bytes”
  - A word is 32 bits or 4 bytes
  - $2^{32}$ bytes = 32 words: addresses 0, 4, 8, ...

Quesion: What are the last two bits in a word address?
Access alignment

- Aligned addresses fall on 4 byte (word) boundaries (e.g., 0, 4, 8, 12...)
- Unaligned addresses do not (e.g., 1, 3, 7, 63)
- Some machines support unaligned accesses (not MIPS)
  - Hardware can convert to multiple aligned accesses (complex)
  - Hardware can detect it and have software fix it (slow)
- But there's typically a performance penalty
  - 2 memory accesses plus merging to get unaligned data
  - Intel provided high performance support in 2010 (Nehalem)

- 8 bits of data

\[
\begin{array}{cccc}
\text{Load Word addr=0} & \text{Aligned} & \text{Aligned} & \text{Aligned} \\
\text{Load Word addr=9} & \text{Not Aligned} & \text{Not Aligned} & \text{Not Aligned}
\end{array}
\]

**Quiz:**
- memory alignment
- Instruction operand orders
- Temporary registers (e.g., \( r5 \leftarrow (r1+r2+r3) \))
### Processor execution model

**Questions:**
- Why do we care about sequential and atomic execution?

**Answer:**
- Without it, we could not understand the program.

- **Sequential**: execute instructions in order.
  - Processor promises that the instruction execution will appear to be sequential and atomic.
  - Processor may not do: $R_3 = R_1 + R_2$ then $R_2 = R_1 + R_2$ ← wrong result

- **Atomic**: execute each instruction all at once.
  - Processor has to finish $R_2 = R_1 + R_2$ before starting $R_3 = R_1 + R_2$

- Processors don’t do either of these things (too slow)
  - But it’s important that they make it look like they do

**Sequen*al**
- Program says: $R_2 = R_1 + R_2$ then $R_3 = R_1 + R_2$

**Atomic**
- Program says: $R_2 = R_1 + R_2$ then $R_3 = R_1 + R_2$

### Stored program computers

**Program and data are stored in memory**
- Instructions have to be fetched from memory for execution
- Data has to be fetched from memory for computation

**No difference between data and instruction memory!**
- This is where a lot of virus attacks come from: writing over data with instructions then executing them
- Buffer overflows

### Example instruction executions

**Data operations**
- Arithmetic (add, multiply, divide)
- Logical (and, or, not, xor)

**Data transfer**
- Load (register ⇒ register)
- Store (register ⇒ memory)

**Sequencing**
- Branch (conditional, e.g.,: Jump (unconditional))

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>add $1, $2, $3</td>
<td>loadw $1, 100($2)</td>
<td>branch $1, $2, $3</td>
</tr>
<tr>
<td>$1 = $2 + $3</td>
<td>memory[$2 + 100] = $1</td>
<td></td>
</tr>
</tbody>
</table>

(Table is printed in your book for reference.)
DATA OPERATIONS

1. Program Counter holds the instruction address.
2. Control logic decodes the instruction and tells the ALU and Register File what to do.
3. ALU executes the instruction and results flow back to the Register File.
4. The Control logic updates the Program Counter for the next instruction.

Add/Sub Example (1 of 2)

- Program Counter:
  - Instruction Address
- ALU (Arithmetic Logic Unit) (Compute)
- Register File
  - R0, R1, R2, R3, R4, R5, R6, R7
- Memory
  - 0, 4, 8, 12, 16, 20, 24, 28, 32

- ALU = Arithmetical Logic Unit (Compute)

Add/Sub Example (2 of 2)

- Program Counter:
  - Instruction Address
- ALU (Arithmetic Logic Unit) (Compute)
- Register File
  - R0, R1, R2, R3, R4, R5, R6, R7
- Memory
  - 0, 4, 8, 12, 16, 20, 24, 28, 32
13/09/2012

Add/sub example (2 of 2)

Program Counter
(PC) fetches the instruction Register.
Control tells the ALU what to do.
ALU writes results back into the Register File.

Practice problem: add/sub

f = (g + h) - (i + j)
R3 = f
R4 = g
R5 = h
R6 = i
R7 = j

ALU

quiz questions:
Increasing the PC who controls which register files to read and write back to
DATA TRANSFER INSTRUCTIONS

Data transfers in detail:
1. ALU generates address.
2. Address goes to the Memory Address Register.
3. Results to/from memory are stored in the Memory Data Register.
4. Data from memory can now be stored back into the Register File or written.

Load word example (1 of 2):
1. addi R5, R0, 12
2. lw R6, R5 123456789
We used the address from the register file to load data from memory.

An offset can be added to addresses as part of the lw/sw instructions.

For stores we need:
the address (from ALU)
the data (from register)
Variable A = 3
Address of A = 24

Program:
Write 512 to A
Put address of A in R5
Put address of R5 in R6

Variable A = 3
Address of A = 24

Program:
Write 512 to A
Put address of A in R5
Put new value of A in R6
Store Mem[R5] ← R6

2. Data transfers
sw R6, 0(R5)

3. Sequencing
addi R5, R0, 24
addi R6, R0, 512
sw R6, R5

Quiz on:
memory address register and data – why do you need two? offsets for loads/stores—where is the offset stored?

SEQUENCING INSTRUCTIONS
Sequencing (control instructions)

- Sequencing instructions make decisions
  - What instruction to execute next?
  - They change the “control flow” of the program

- MIPS conditional branch instructions
  - bne R0, R1, Label branch if not equal to label
  - beq R5, R4, Label branch if equal to label

- Example:
  \[
  R1 = i; R2 = j; R3 = h
  \]

  
  if \((i=j)\)
  \[
  bne R1, R2, Skip
  \]

  \[
  h = i+j; \quad \text{add} R3, R1, R2
  \]

  \[
  \ldots \quad \text{Skip; } \ldots
  \]

  
  \[
  \text{Branch to here if } R1 \neq R2
  \]

Sequencing: unconditional jump

- MIPS unconditional branch instruction: jump
  - \(j \text{ label}\) jump to label

- Example:
  \[
  R1 = i; R2 = j; R3 = h
  \]

  
  if \((i=j)\)
  \[
  bne R1, R2, Skip
  \]

  \[
  h = i+j; \quad \text{add} R3, R1, R2
  \]

  \[
  \ldots \quad \text{Skip; } \ldots
  \]

  
  \[
  \text{Branch to here if } R1 = R2
  \]

  \[
  \text{else}
  \]

  \[
  h = i-j; \quad \text{sub} R3, R1, R2
  \]

  \[
  \ldots \quad \text{SkipSub; } \ldots
  \]

  
  \[
  \text{Always skip the subtraction if we did the addition.}
  \]

Branch Instructions

- Change the flow of the program \(\rightarrow\) change the Program Counter
  - \(j\) jump whatever
  - \(beq\) branch if equal
  - \(bne\) branch if not equal

- Example: if \((a=b)\) \(c=1;\) else \(c=2;\)

  \[
  R5 = a; R6 = b; R7 = c
  \]

  
  if \((a=b)\)
  \[
  \text{bne R5, R6, 12; } \text{if } (R5 \neq R6) \text{ goto 12}
  \]

  \[
  c=1; \quad 4 \quad \text{add} R7, R6, 1
  \]

  \[
  0 \quad \text{j 16}
  \]

  \[
  0 \quad \text{goto 16}
  \]

  \[
  \text{else } c=2; \quad 12 \quad \text{add} R7, R5, 2
  \]

  \[
  0 \quad \text{R7} \leftarrow 2+0
  \]

  
  \[
  \text{Always skip setting to 2 if we set it to 1.}
  \]
Sequencing: Loops

for (j=0; j<10; j++) {
    b = b + j;
    ...
}

R5 = j; R6 = b;

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>add R5, R0, 0</td>
<td>j = 0 + 0</td>
</tr>
<tr>
<td>4</td>
<td>add R1, R0, 10</td>
<td>R1 = j + 10</td>
</tr>
<tr>
<td>8</td>
<td>beq R5, R1, 24</td>
<td>if (j == 10) goto 24</td>
</tr>
<tr>
<td>12</td>
<td>add R6, R6, R5</td>
<td>b = b + j</td>
</tr>
<tr>
<td>16</td>
<td>add R5, R5, 1</td>
<td>j = j + 1</td>
</tr>
<tr>
<td>20</td>
<td>j 0</td>
<td>goto 8</td>
</tr>
<tr>
<td>24</td>
<td>...</td>
<td>pop out of loop, continue</td>
</tr>
</tbody>
</table>

We need the constant 10 for the loop comparison, so put it in R1.

Sequencing in detail

The label constant in instruction words, as it needs to be compared by the constant in bytes.

3. Sequencing

Quiz: conditional vs. unconditional needing both to do if/else how to check ==7