Computer Systems Architecture

http://cs.nott.ac.uk/~txa/g51csa/

Thorsten Altenkirch and Liyang Hu

School of Computer Science and IT
University of Nottingham

Lecture 04: Machine Code, Data Transfer and Control Flow
Assembly to Machine Code

- So far we've been using assembly language
- Assembler turns *symbolic instructions* to *machine code*
- Each instruction is 32-bits (or 4 bytes) long
- The 32 bit machine code consists of several fields
  - Field format differ depending on instruction type
  - MIPS uses three basic formats: R, I and J

<table>
<thead>
<tr>
<th>Assembly</th>
<th>add $s0, $s1, $s2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal</td>
<td>$0232\ 8020_{16}$</td>
</tr>
<tr>
<td>Binary</td>
<td>$0000\ 0010\ 0011\ 0010\ 1000\ 0000\ 0010\ 0000_{2}$</td>
</tr>
</tbody>
</table>
# Instruction Encoding (Register Operands)

## R-Format

<table>
<thead>
<tr>
<th>Field</th>
<th>op</th>
<th>src₀</th>
<th>src₁</th>
<th>dst</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

→ 32 bits total

- **op**: Basic operation code
- **src**: Source operand register
- **dst**: Destination operand register
- **shamt**: Shift amount (not relevant now; more later)
- **func**: Function / operation variant
Instruction Encoding (Immediate Operand)

I-Format

<table>
<thead>
<tr>
<th>Field</th>
<th>op</th>
<th>src</th>
<th>dst</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

32 bits total

- **op**: Basic operation code
- **src**: Source operand register
- **dst**: Destination operand register
- **imm**: Immediate constant

Also a J format: 6 bit opcode, 26-bit immediate
## Encoding Addition

**add $s0, $s1, $s2 – 02328020_{16}**

<table>
<thead>
<tr>
<th></th>
<th>$src_0$</th>
<th>$src_1$</th>
<th>$dst$</th>
<th>$shamt$</th>
<th>$func$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$op$</td>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>10000</td>
<td>00000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$src$</th>
<th>$dst$</th>
<th>$imm$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$addi$</td>
<td>$s1$</td>
<td>$s2$</td>
<td>$s0$</td>
</tr>
<tr>
<td>$addi$</td>
<td>$t1$</td>
<td>$t0$</td>
<td>42</td>
</tr>
</tbody>
</table>

**addi $t0, $t1, 42 – 2128002A_{16}**

<table>
<thead>
<tr>
<th></th>
<th>$src$</th>
<th>$dst$</th>
<th>$imm$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$op$</td>
<td>001000</td>
<td>01001</td>
<td>01000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$src$</th>
<th>$dst$</th>
<th>$imm$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$addi$</td>
<td>$t1$</td>
<td>$t0$</td>
<td>42</td>
</tr>
</tbody>
</table>
Registers vs Memory

- Processor can access registers directly
- Limit of 32 registers
  - Most programs require much more data
- What about larger data structures?
  - Document in a text editor
  - Program code
  - Graphics image displayed on screen
- Extra data must be kept in memory

  **load** transfer data from memory to register
  **store** transfer data from register to memory
Load Word

lw \( dst, \ n(src) \)

- Load word at address \( src \) offset \( n \) into \( dst \)
- \( dst := M[src + n] \)
- \( M[addr] \) – word/half/byte at address \( addr \)
- Offset \( n \) is 16 bits (\( lw \) is an l-format instruction)
  - Can write \( lw \ dst, (src) \) if \( n \) is zero
- The address \( src + n \) must be word-aligned
  - i.e. divisible by 4
**Store Word**

\[ \text{sw} \ dst, \ n(src) \]

- Stores word in \( dst \) into address \( src \) offset \( n \)
- \( M[src + n] := dst \)
- Note \( dst \) is not the ‘destination’ this time
- Offset \( n \) is 16 bits (\text{sw} is an I-format instruction)
- Again, the address \( src + n \) must be *word-aligned*
Example: Loads and Stores

**Machine Code**

- **Before**
  - Address | Data
  - 10010000 | 7C0802A6\textsubscript{16}
  - 10010004 | BE81FFD0\textsubscript{16}

- **After**
  - Address | Data
  - 10010000 | BE81FFD0\textsubscript{16}
  - 10010004 | 7C0802A6\textsubscript{16}

**Assembly Code**

- Initially $s0 = 10010000\textsubscript{16}$
  - lw $t0, ($s0)
  - lw $t1, 4($s0)
  - sw $t0, 4($s0)
  - sw $t1, ($s0)

- Afterwards,
  - $t0 = 7C0802A6\textsubscript{16}$
  - $t1 = BE81FFD0\textsubscript{16}$
Bytes

- Not all data is word-sized, e.g. ASCII characters
- No alignment requirements
  - Addresses are always byte-aligned

Byte Transfer Instructions

- `lbu dst, n(src)` – load byte unsigned
  - Loads $M[src + n]$ into the lower 8 bits of $dst$
  - Sets remaining 24 bits of $dst$ to zero
  - Ignore ‘unsigned’ for now; more later
  - There’s also `lb dst, n(src)` – load byte

- `sb dst, n(src)` – store byte
  - Stores bits 0 to 7 of $dst$ into $M[src + n]$
  - Ignores remaining 24 bits of $dst$
Half-Words

- Half-words are 16 bits long (i.e. 2 bytes)
- Address $src + n$ must be half-word aligned
  - Sum of $src + n$ must be even

Half-Word Transfer Instructions

- **lhu** $dst, n(src)$ – load half-word unsigned
  - Loads $M[src + n]$ into the lower 16 bits of $dst$
  - Sets remaining 16 bits of $dst$ to zero
  - Ignore ‘unsigned’ for now; more later
  - There’s also **lh** $dst, n(src)$ – load half-word

- **sh** $dst, n(src)$ – store half-word
  - Stores bits 0 to 15 of $dst$ into $M[src + n]$
  - Ignores remaining 16 bits of $dst$
What does this program do?

.data
num: .word 0

.text
.globl main
main: la $s0, num
li $v0, 5 # read_int
syscall

sw $v0, ($s0)
lbu $a0, 1($s0)

li $v0, 1 # print_int
syscall
j $ra
High-Level Control Structures

- Unstructured programming leads to ‘spaghetti code’
- The goto keyword is avoided in high-level programming
  - Reserved keyword in Java, but does nothing
  - Can be used for jumps in C, C++, and C#.
- Instead we use control structures like
  - if-then-else
  - while and for, break and continue
- Edsgar Dijkstra’s “Goto Statement Considered Harmful”
  Communications of the ACM, March 1968
  http://www.acm.org/classics/oct95/
  - See also http://en.wikipedia.org/wiki/GOTO or
    http://reference.com/browse/wiki/GOTO
Control Flow in Assembly

- No high-level control structures in assembly
- *Branch* (or jump) instructions change the PC
  - Hence affects the next instruction to be executed
  - The sequence of branches in a program is its *control flow*
- *Conditional* branches depend on a previous comparison
  - Otherwise we carry on with the following instruction
- *Unconditional* branches always jumps to the given location
Basic Branching

**j label** – jump (unconditional branch)
- Continues execution at label — PC := [label]

**beq dst, src, label** – branch on equal
- Continues execution at label if dst = src
- if(dst == src) PC := [label]

**bne dst, src, label** – branch on not-equal
- Continues execution at label if dst ≠ src
- if(dst != src) PC := [label]

Labels identify addresses of program code as well as data.
If-Then-Else

- How do we implement the following Java fragment?

```
if(i == j)
    f = g + h;
else
    f = g - h;
```

- Assume

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s0$</td>
<td>$s1$</td>
<td>$s2$</td>
<td>$s3$</td>
<td>$s4$</td>
</tr>
</tbody>
</table>
Implementing Decisions

```
bne $s3, $s4, else_i_ne_j
add $s0, $s1, $s2  # then-block
j if_i_eq_j_end
else_i_ne_j:
    sub $s0, $s1, $s2  # else-block
if_i_eq_j_end:

- Branch on opposite condition (i ≠ j) to the else-block
- So if i = j, fall through to the add instruction
- Last instruction of if-block jumps over the else-block
- What if there is no else-block?
```
Example: Checking for a Passcode

```assembly
# ...
main: li $v0, 5 # read_int
syscall

li $t0, 42      # secret reply
beq $v0, $t0, correct
la $a0, go_away
j check_end

correct:
    la $a0, hello

check_end:

li $v0, 4 # print_string
syscall
j $ra
```
Thursday quiz

Most of the following questions are multiple choice. There is at least one correct choice but there may be several. For each of the questions list all the roman numerals corresponding to correct answers but none of the incorrect ones. Questions are marked as follows:

- **no errors** 5 points
- **1 error** 3 points
- **2 errors** 1 point
- **≥3 errors** 0 points
Thursday quiz

1. What are important aspects of the von Neumann architecture?
   
   a. Instructions are data.
   b. Separate memory for programs and data.
   c. Invented in Germany.
   d. Uses a Program Counter (PC).
   e. Superseded by the Harvard Architecture.
Thursday quiz

2. What is MIPS?
   a. Multiple Instructions per step
   b. Microprocessor without Interlocked Processor States
   c. Microprocessor with Interlocked Processor States
   d. A typical RISC processor
   e. A typical CISC processor
3. What are typical features of assembly language?

- Universal language for different processors
- Processor specific
- Uses labels for memory addresses
- Uses hexadecimal notation for instructions
- Is a compromise between C and Java.
Thursday quiz

4. What can we say about MIPS’ registers?
   a. Designed for different purposes
   b. 32 registers because the word size is 32.
   c. Saved registers are not changed by system calls
   d. Temporary registers are not changed by system calls
   e. Register 0 always contains 0.
5. What is the effect of add $s1, $s1, $s0?

a. Stores the sum of $s1 and $s0 in $s0.
b. Stores the sum of $s1 and $s1 in $s0.
c. Stores the sum of $s1 and $s0 in $s1.
d. Causes an interrupt because $s1 is used twice.
e. $s1 remains unchanged because $s0 always contains 0.