LECTURE 4: ASSEMBLY
April 7, 2011

Levels of Representation

Compiler
High Level Language Program (e.g., C)

Assembler
Assembly Language Program (e.g., MIPS)

Machine Language Program (MIPS)

Machine Interpretation
Hardware Architecture Description (Logic, Logisim, etc.)

Architecture Implementation
Logic Circuit Description (Logisim, etc.)

- temp = v[k];
- v[k] = v[k+1];
- v[k+1] = temp;

- lw $t0, 0($2)
- lw $t1, 4($2)
- sw $t1, 0($2)
- sw $t0, 4($2)
MIPS Instruction Set

- Each instruction executes exactly one of a short list of simple commands
- Instructions are divided into three kinds of format (R, I and J format)
  - Common arithmetic instructions (R format)
  - Memory load and store (I format)
  - Branching and jump instructions (J format)

Your First MIPS Instructions!

- All R instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: \( A = B + C \)
MIPS code: `add $s0, $s1, $s2`

C code: \( A = B - C \)
MIPS code: `sub $s0, $s1, $s2`

\( A, B, C \) in C programming are variables
\( S0-S2 \) in Assembly are registers
Compiler associates variables with registers
Registers (1/4)

- Unlike high level language like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Arithmetic (aka R) instructions operands must be registers
  - limited number of special locations built directly into the hardware
  - operations can only be performed on these!
- Compiler associates variables with registers

Registers (2/4)

- Benefit: Since registers are directly in hardware, they are very fast (faster than 1 billionth of a second)
- Drawback: Since registers are in hardware, there are a predetermined number of them
  - Solution: MIPS code must be very carefully put together to efficiently use registers
Registers (3/4)

- Each MIPS register is **32** bits wide
- Groups of **32** bits called a **word** in MIPS
- **32** registers in MIPS
  - Why 32? **Smaller is faster**
  - Registers are numbered from 0 to 31
- Each register can be referred to by number or name
  - Number references:
    - $0, $1, $2, ... $30, $31
  - By convention, each register also has a name to make it easier to code

Registers (4/4)

Use names to make your code more readable

<table>
<thead>
<tr>
<th>Register name</th>
<th>Common name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16-$23</td>
<td>$s0-$s7</td>
<td>Saved registers to use freely. Preserved across function calls. (correspond to C variables)</td>
</tr>
<tr>
<td>$8 -$15</td>
<td>$t0-$t7</td>
<td>Temporary registers you can use as you want. Not guaranteed to be preserved across function calls.</td>
</tr>
<tr>
<td>$0</td>
<td>$zero</td>
<td>Hard wired to the value zero, always has the value 0. Any writes to this register are ignored.</td>
</tr>
<tr>
<td>$1,$26, $27,$29</td>
<td></td>
<td>Reserved for special purposes by the assembler, compiler and operating system.</td>
</tr>
</tbody>
</table>

Others will be introduced later
Register Zero

- One particular immediate, the number zero (0), appears very often in code.
- So we define register zero ($0 or $zero) to always have the value 0;
- **add $s0,$s1,$zero** (in MIPS)
  \[ f = g \text{ (in C)} \]
  where MIPS registers $s0,$s1 are associated with C variables f, g

Immediates

- Immediates are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:
  **addi $s0,$s1,10** (in MIPS)  \[ f = g + 10 \text{ (in C)} \]
  where MIPS registers $s0,$s1 are associated with C variables f, g
- Syntax similar to add instruction, except that last argument is a number instead of a register.
- Is there any subi?
Addition and Subtraction of Integers

• How do we do this?
  \[ f = (g + h) - (i + 10); \]
  where MIPS registers $s0, s1, s2, s3$ are associated with C variables \( f, g, h, i \)

• Use intermediate temporary register
  \[
  \begin{align*}
  \text{add} & \quad \text{add} \quad \text{add} \quad \text{add} \\
  $t0, s1, s2$ & \quad \text{d, s, t} \quad \text{d, s, s, t} \quad \text{d, s, s, t} \quad \text{d, s, t} \\
  \text{addi} & \quad \text{addi} \quad \text{addi} \quad \text{addi} \\
  $t1, s3, 10$ & \quad \text{d, s, imm} \quad \text{d, s, imm} \quad \text{d, s, imm} \\
  \text{sub} & \quad \text{sub} \quad \text{sub} \quad \text{sub} \\
  $s0, t0, t1$ & \quad \text{d, s, t} \quad \text{d, s, t} \quad \text{d, s, t} \\
  \end{align*}
  \]
  \[ f = (g + h) - (i + 10) \]
Overflow in Arithmetic

- Some languages detect overflow (Ada), some don’t (C)
- MIPS solution is 2 kinds of arithmetic instructions to recognize 2 choices:
  - add (add), add immediate (addi) and subtract (sub) cause overflow to be detected
  - add unsigned (addu), add immediate unsigned (addiu) and subtract unsigned (subu) do not cause overflow detection
- Compiler selects appropriate arithmetic
  - MIPS C compilers produce addu, addiu, subu

MIPS Arithmetic

<table>
<thead>
<tr>
<th>Instruction</th>
<th>C code</th>
<th>MIPS code</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$A = B + C$</td>
<td>add $s0$, $s1$, $s2$</td>
</tr>
<tr>
<td>subtract</td>
<td>$A = B - C$</td>
<td>sub $s0$, $s1$, $s2$</td>
</tr>
</tbody>
</table>

- $A = B + C + D$
- $A = B + C - D$
- $A = (B+C) - (D+E)$
Multiplication

- In MIPS, we multiply registers, so:
  - 32-bit value x 32-bit value = 64-bit value

- Syntax of Multiplication (signed):
  - mult register1, register2
  - Multiplies 32-bit values in those registers & puts 64-bit product in special result regs:
    - puts product upper half in hi, lower half in lo
  - hi and lo are 2 registers separate from the 32 general purpose registers
  - Use mfhi register & mflo register to move from hi, lo to another register

Multiplication Example

- in C: a = b * c;
- in MIPS:
  - let b be $s2; let c be $s3; and let a be $s0 and $s1 (since it may be up to 64 bits)
    - mult $s2,$s3  # b*c
    - mfhi $s0  # upper half of
      - # product into $s0
    - mflo $s1  # lower half of
      - # product into $s1
  - Note: Often, we only care about the lower half of the product.
Division

• Syntax of Division (signed):
  • `div register1, register2`
  • Divides 32-bit register 1 by 32-bit register 2:
    • puts remainder of division in `hi`, quotient in `lo`

• Example in C:
  a = c / d;
  b = c % d;

• in MIPS: `a←$s0; b←$s1; c←$s2; d←$s3`
  `div $s2,$s3` # `lo=c/d, hi=c%d`
  `mflo $s0` # get quotient
  `mfhi $s1` # get remainder

ASSEMBLY LANGUAGE

So far: It is like a calculator
add, sub, addi
multi, div
Assembly Operands: Memory

- C variables map onto registers; what about large data structures like arrays?
- Memory contains such data structures
- But MIPS arithmetic instructions only operate on registers, never directly on memory.
- **Data transfer instructions** transfer data between registers and memory:
  - Memory to register
  - Register to memory

5 components of any Computer

Registers are in the datapath of the processor; if operands are in memory, we must transfer them to the processor to operate on them, and then transfer back to memory when done.

These are “data transfer” instructions...
MIPS MEMORY 101

Addressing: Byte vs. word

• Every word in memory has an address, similar to an index in an array
• Early computers numbered words like C numbers elements of an array:
  • Memory[0], Memory[1], Memory[2], ...

Called the “address” of a word
• Computers needed to access 8-bit bytes as well as words (4 bytes/word)
• Today machines address memory as bytes, (i.e., “Byte Addressed”) hence 32-bit (4 byte) word addresses differ by 4
  • Memory[0], Memory[4], Memory[8], ...
MIPS Byte Addressing

- MIPS requires that all words start at byte addresses that are multiples of 4 bytes.

Memory Alignment

- Last hex digit of address is:
  - 0, 4, 8, or $C_{\text{hex}}$
  - 1, 5, 9, or $D_{\text{hex}}$
  - 2, 6, A, or $E_{\text{hex}}$
  - 3, 7, B, or $F_{\text{hex}}$

- Called **Alignment**: objects must fall on address that is multiple of their size.
Role of Registers vs. Memory

- What if more variables than registers?
  - Compiler tries to keep most frequently used variable in registers
  - Less common in memory: spilling
- Why not keep all variables in memory?
  - Smaller is faster: registers are faster than memory
  - Registers more versatile:
    - MIPS arithmetic instructions can read 2, operate on them, and write 1 per instruction
    - MIPS data transfer only read or write 1 operand per instruction, and no operation

BACK TO MIPS MEMORY ACCESS

Load & Store
Data Transfer: Memory to Reg

To transfer a word of data, we need to specify two things:
1. Register: specify this by # ($0 - $31) or symbolic name ($s0, …, $t0, …)
2. Memory address: more difficult
   - Think of memory as a single one-dimensional array, so we can address it simply by supplying a pointer to a memory address.
   - Other times, we want to be able to offset from this pointer.
   - Remember: “Load FROM memory”

---

Data Transfer: Memory to Reg

- To specify a memory address to copy from, specify two things:
  - A register containing a pointer to memory
  - A numerical offset (in bytes)
- The desired memory address is the sum of these two values.
- Example: $8 ($t0)
  - specifies the memory address pointed to by the value in $t0, plus 8 bytes
Data Transfer: Memory to Reg

- Load Instruction Syntax:
  
  1  2,3(4)

  where

  1) operation name
  2) register that will receive value
  3) numerical offset in bytes
  4) register containing pointer to memory

- MIPS Instruction Name:
  
  lw (meaning Load Word, so 32 bits or one word are loaded at a time)

Example:

```
lw $t0,12($s0)
```

This instruction will take the pointer in $s0, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register $t0.

Notes:

- $s0 is called the base register
- 12 is called the offset
- offset is generally used in accessing elements of array or structure; base reg points to beginning of array or structure
Data Transfer: Reg to Memory

- Also want to store from register into memory
  - Store instruction syntax is identical to Load’s
- MIPS Instruction Name:
  - `sw` (meaning Store Word, so 32 bits or one word are loaded at a time)

  Data flow

- Example: `sw $t0,12($s0)`
  - This instruction will take the pointer in $s0, add 12 bytes to it, and then store the value from register $t0 into that memory address
  - Remember: “Store INTO memory”

Pointers v. Values

- Key Concept: A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory address), and so on
- If you write `add $t2,$t1,$t0`
  - then $t0 and $t1 better contain values
- If you write `lw $t2,0($t0)`
  - then $t0 better contain a pointer
- Don’t mix these up!
Compilation with Memory

- 4x5=20 to select A[5]: byte v. word
- Compile by hand using registers:
  \[ g = h + A[5]; \]
  - g: $s1, h: $s2, base address of A: $s3
- 1st transfer from memory to register:
  - `lw $t0,20($s3)` # $t0 gets A[5]
  - Add 20 to $s3 to select A[5], put into $t0
  - Next add it to h and place in g
    `add $s1,$s2,$t0` # $s1 = h+A[5]

Comments in Assembly

- Another way to make your code more readable: comments!
- Hash (#) is used for MIPS comments
  - anything from hash mark to end of line is a comment and will be ignored
- Note: Different from C.
  - C comments have format
    ```c
    /* comment */
    ```
    so they can span many lines
Notes about Memory

• Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.
• Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes.
• So remember that for both lw and sw, the sum of the base address and the offset must be a multiple of 4 (to be word aligned)

Additional notes

• The store word instruction, sw, copies data from a register to memory. The register is not changed. The memory address is specified using a base/register pair.
  • sw t,off(b) # Word at memory address (b+off) <-- $t
    # b is a register.
    # off is 16-bit two's complement.
• As with the lw instruction, the memory address must be word aligned (a multiple of four).
• Format:
  • sw $12 , 0xFFF8($13)
  • sw $12 , -8($13)
Symbolic Address

```assembly
## poly.asm
## ## evaluate 5x^2 -12x + 97
##
.text
.globl main
main:
  . . . many instructions
.data
# In SPIM, the data section
# starts at address 0x10000000
x: .word 17
  # The base register points here.
poly: .word 0
## End of file
```

.data:The start of the data section of memory.
.word means: put a 32-bit two's complement integer here. The integer is specified using base 10 (by default).

The .word 17 calls for a 32-bit two's complement representation of an integer that in base 10 is "17". The assembler converts the representation into the appropriate bit pattern. Can also specify the bit pattern using the hexadecimal name for the pattern. .word 0x11 does exactly the same thing as .word 17.

So Far...

- All instructions have allowed us to manipulate data.
- So we’ve built a calculator.
- In order to build a computer, we need ability to make decisions…