CS61C – Machine Structures

Lecture 8 - Introduction to the MIPS Processor and Assembly Language

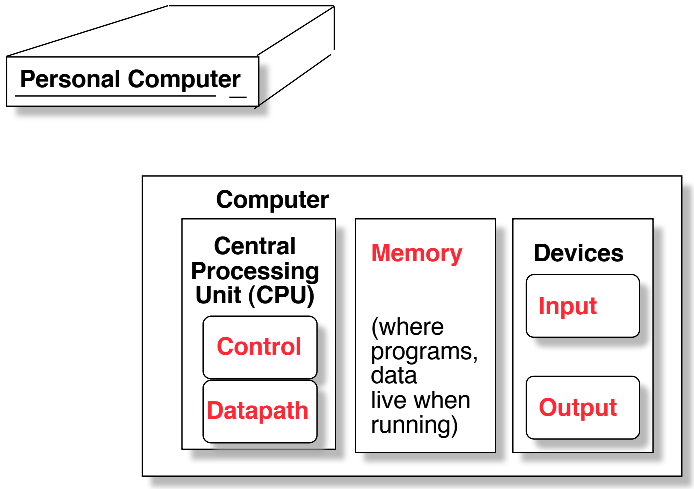
2/3/2006

John Wawrzynek

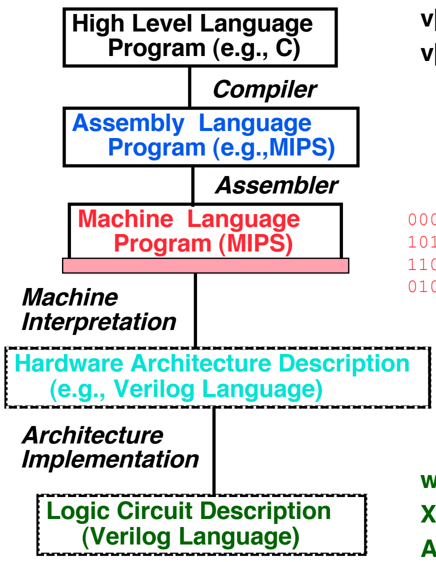
(www.cs.berkeley.edu/~johnw)

www-inst.eecs.berkeley.edu/~cs61c/

Anatomy: 5 components of any Computer


Personal Computer
Computer
Central Processing Unit (CPU)
Control
Datapath
Memory
(where programs, data live when running)
Devices
Input
Output

61C Levels of Representation


High Level Language Program (e.g., C)
Compiler
Assembly Language Program (e.g.,MIPS)
Assembler
Machine Language Program (MIPS)
Machine Interpretation
Hardware Architecture Description (e.g., Verilog Language)
Architecture Implementation
Logic Circuit Description (Verilog Language)

```
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)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

```
wire [31:0] dataBus;
regFile registers (databus);
ALU ALUBlock (inA, inB, databus);
```

```
wire w0;
XOR (w0, a, b);
AND (s, w0, a);
```

Assembly Language

° "Instructions" are the primitive operations that a CPU may execute.

° Different CPUs implement slightly different sets of instructions. The set of instructions a particular CPU implements is called an *Instruction Set Architecture* (*ISA*).

  • Examples: Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS, Intel IA64, ARM, ...

° Assembly language is a textual version of these instructions.

° Assembly language is used:

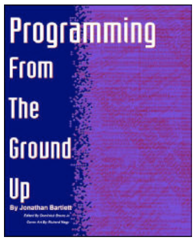
  • As an output of the C compiler, or

  • As a means to directly program the CPU. Why would anyone want to do this?!

Book: *Programming From the Ground Up*

"A new book was just released which is based on a new concept - teaching computer science through assembly language (Linux x86 assembly language, to be exact). This book teaches how the machine itself operates, rather than just the language. I've found that the key difference between mediocre and excellent programmers is whether or not they know assembly language. ***Those that do tend to understand computers themselves at a much deeper level.*** *Although [almost!] unheard of today, this concept isn't really all that new -- there used to not be much choice in years past. Apple computers came with only BASIC and assembly language, and there were books available on assembly language for kids. This is why the old-timers are often viewed as 'wizards': they* ***had*** *to know assembly language programming."*

-- `slashdot.org` comment, 2004-02-05



Instruction Set Architectures

- **Early trend was to add more and more instructions to new CPUs to do elaborate operations**
  - **VAX architecture had an instruction to evaluate polynomials!**
- **RISC philosophy (Cocke IBM, Patterson, 1980s) – Reduced Instruction Set Computing**
  - **Keep the instruction set small and simple, makes it easier to build fast hardware.**
  - **Let software do complicated operations by composing simpler ones.**

MIPS Architecture

- MIPS – semiconductor company that built one of the first commercial RISC architectures
- We will study the MIPS architecture in some detail in this class (also used in upper division courses CS 152, 162, 164)
- Why MIPS instead of Intel 80x86?
  - MIPS is simple, elegant. Don't want to get bogged down in gritty details.
  - MIPS widely used in embedded apps.

Most HP LaserJet workgroup printers are driven by MIPS-based™ 64-bit processors.

Assembly Variables: Registers (1/4)

- Unlike HLL like C or Java, MIPS assembly cannot operate directly on variables in memory
  - Why not? Keep Hardware Simple
- Assembly Operands are registers
  - limited number of special locations built directly into the CPU
  - operations can only be performed on these!
- Benefit: Since registers are directly in the CPU, they are very fast (faster than 1 billionth of a second)

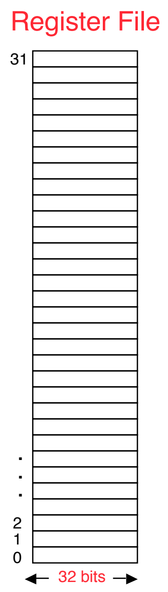
Assembly Variables: Registers (2/4)

- Drawback: Since registers are in the CPU and must be fast, there are a limited number of them
  - Solution: MIPS code must be very carefully put together to efficiently use registers
- MIPS has 32 registers
  - Why only 32?
    - Smaller is faster
    - Small register addresses keeps instruction representation small.
- Each MIPS register is 32 bits wide
  - Groups of 32 bits called a word in MIPS

Assembly Variables: Registers (3/4)

- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- Number references:

```
$0, $1, $2, … $30, $31
```



Assembly Variables: Registers (4/4)

- By convention, each register also has a name to make it easier to code
- For now:

  `$16 - $23` ➔ `$s0 - $s7`

  (correspond to C variables)

  `$8 - $15` ➔ `$t0 - $t7`

  (correspond to temporary variables)

  Later will explain other 16 register names
- In general, use names to make your code more readable

C, Java variables vs. registers

- In C (and most High Level Languages) variables declared first and given a type
  - Example:

    ```
    int fahr, celsius;
    char a, b, c, d, e;
    ```
- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match `int` and `char` variables).
- In Assembly Language, the registers have no type; operation determines how register contents are treated

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
    `/* comment */`
    so they can span many lines

Assembly Instructions

° In assembly language, each statement (called an Instruction), executes exactly one of a short list of simple commands

° Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction

° Instructions are related to operations (=, +, -, *, /) in C or Java

° Ok, enough already…gimme my MIPS!

MIPS Addition and Subtraction (1/4)

° Syntax of Instructions:

1 2,3,4

where:

1) operation by name

2) operand getting result (“destination”)

3) First operand for operation (“source1”)

4) Second operand for operation (“source2”)

° Syntax is rigid:

- 1 operator, 3 operands
- Why? Keep Hardware simple via regularity

Addition and Subtraction of Integers (2/4)

° Addition in Assembly

- Example: `add $s0,$s1,$s2` (in MIPS)

  Equivalent to: `a = b + c` (in C)

where MIPS registers `$s0,$s1,$s2` are associated with C variables `a, b, c`

° Subtraction in Assembly

- Example: `sub $s3,$s4,$s5` (in MIPS)

  Equivalent to: `d = e - f` (in C)

where MIPS registers `$s3,$s4,$s5` are associated with C variables `d, e, f`

Addition and Subtraction of Integers (3/4)

- How do the following C statement?

```
a = b + c + d - e;
```

- Break into multiple instructions

```
add $t0, $s1, $s2 # temp = b + c
add $t0, $t0, $s3 # temp = temp + d
sub $s0, $t0, $s4 # a = temp - e
```

- Notice: A single line of C may break up into several lines of MIPS.
- Notice: Everything after the hash mark on each line is ignored (comments)

Addition and Subtraction of Integers (4/4)

- How do we do this?

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

- Use intermediate temporary register

```
add $t0,$s1,$s2     # temp = g + h
add $t1,$s3,$s4     # temp = i + j
sub $s0,$t0,$t1     # f=(g+h)-(i+j)
```

Register Zero

- The number zero (0), appears very often in code.
- MIPS defines register zero (`$0` or `$zero`) to always have the value 0; eg

  `add $s0,$s1,$zero` (in MIPS)

  `f = g` (in C)

  where MIPS registers `$s0,$s1` are associated with C variables `f, g`
- $0 is unchangeable, so an instruction

  `add $zero,$zero,$s0`

  will not do anything!

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` (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.

Immediates

- There is no Subtract Immediate in MIPS: Why?
- Limit types of operations that can be done to absolute minimum
  - if an operation can be decomposed into a simpler operation, don't include it
  - `addi ..., -X` = `subi ..., X` => so no `subi`
- `addi $s0,$s1,-10` (in MIPS)

  `f = g - 10` (in C)

  where MIPS registers `$s0,$s1` are associated with C variables `f, g`

Quiz:

True or False

A. Types are associated with declaration in C (normally), but are associated with instruction (operator) in MIPS.

B. Since there are only 8 local (`$s`) and 8 temp (`$t`) variables, we can't write MIPS for C exprs that contain > 16 vars.

C. If `p` (stored in `$s0`) were a pointer to an array of `int`s, then `p++;` would be `addi $s0 $s0 1`

“And in Conclusion…”

- In MIPS Assembly Language:
  - Registers replace C variables
  - One Instruction (simple operation) per line
  - Simpler is Better
  - Smaller is Faster
- New Instructions:

  `add, addi, sub`
- New Registers:

  C Variables: `$s0` - `$s7`

  Temporary Variables: `$t0` - `$t9`

  Zero: `$zero`