Instructions in Machine Language

Andrea Janes

Content

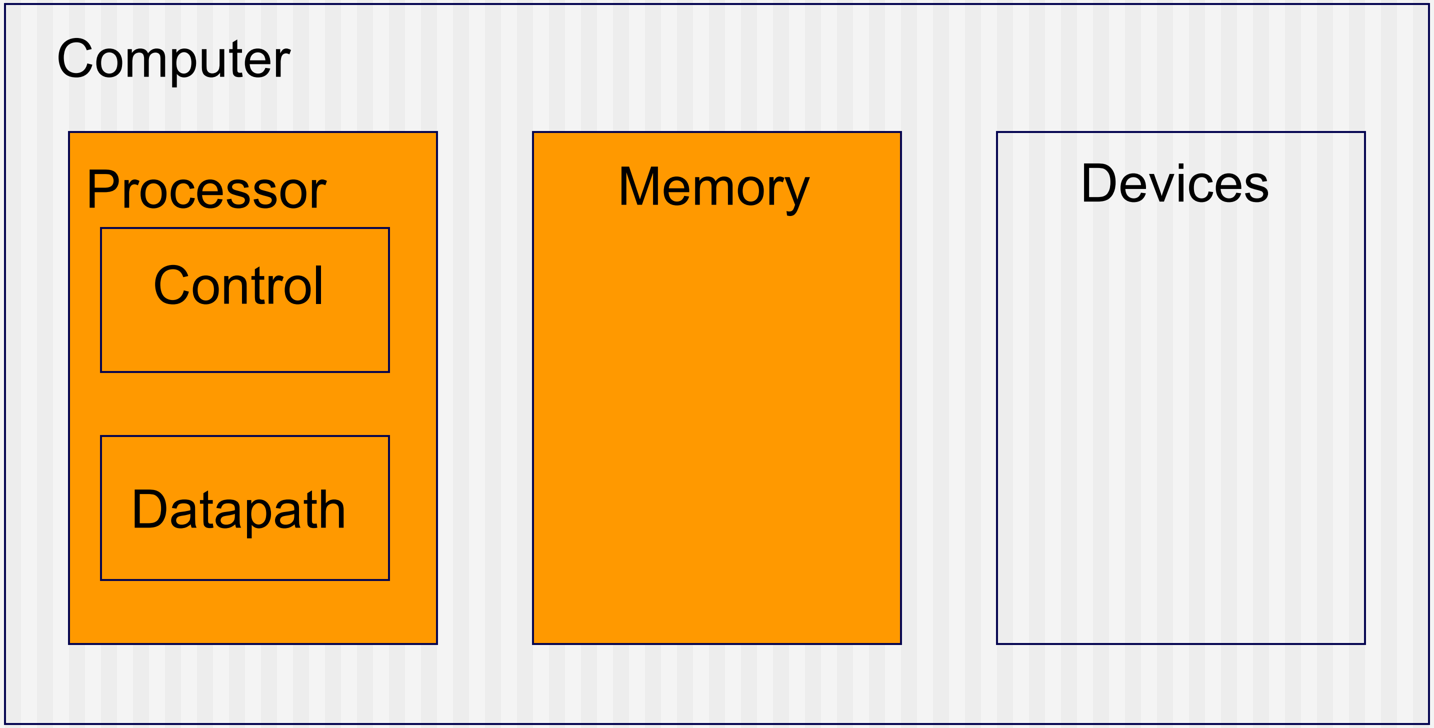
- **Review**
- Again on the Stored Program concept
- Program preparation and structure of the instructions
- Simple math operations

Review: the Instruction set

Machine specific language:

- **Lowest level of programming**
- **Read and executed by the CU**
- **Ex. add R1, R2, R3**
- **Instruction set used here: MIPS**

Review: 5 components

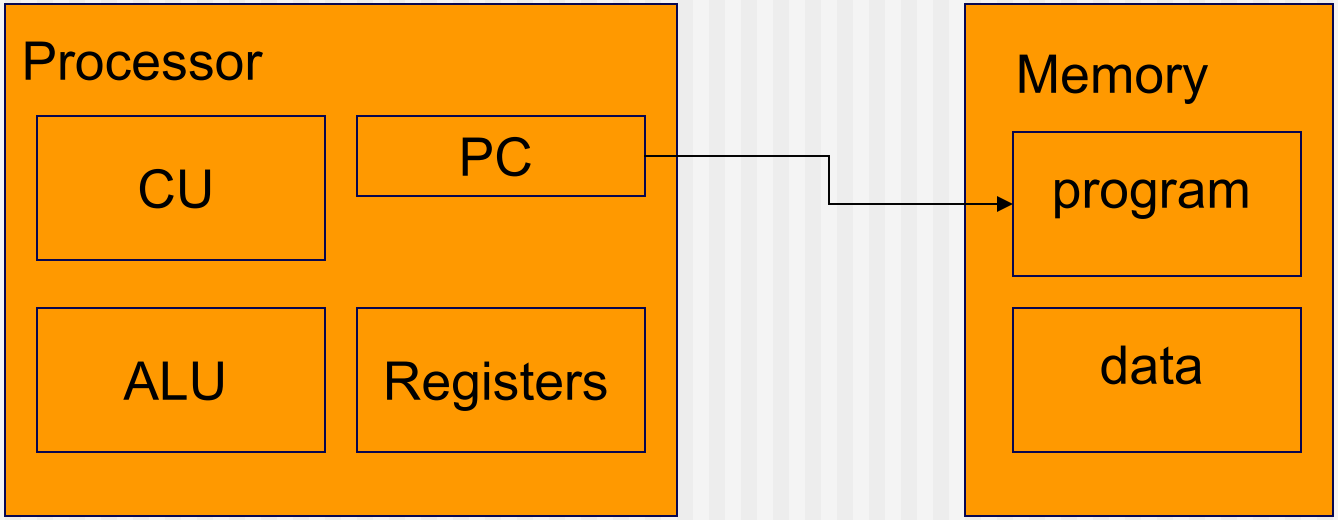


More in detail…

Content

- *Review*
- **Again on the Stored Program concept**
- Program preparation and structure of the instructions
- Simple math operations

Stored program concept



Program execution: step by step of machine language instructions

The Stored-Program Concept

- Computers built on 2 key principles:
  1) Instructions are represented as numbers.
  2) Therefore, entire programs can be stored in memory to be read or written just like numbers (data).
- Simplifies SW/HW of computer systems:
  - Memory technology for data also used for programs

Consequence #1: Everything Addressed

- Since all instructions and data are stored in memory as numbers, everything has a memory address: instructions, data words
  - *both branches and jumps use these*
- Java references are just memory addresses: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- One register keeps address of instruction being executed: "**Program Counter**" (PC)
  - Basically a pointer to memory: Intel calls it Instruction Address Pointer, which is better

Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for Macintosh and IBM PC
- New machines want to run old programs (“binaries”) as well as programs compiled to new instructions
- Leads to instruction set evolving over time
- Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium III); could still run program from 1981 PC today

Content

- *Review*
- *Again on the Stored Program concept*
- **Program preparation and structure of the instructions**
- Simple math operations

Program preparation: compiler

C-program
Swap (int v[], int k)
...

compiler

Machine language program
0010000010100011
1011110110101111
...

Compiler decides:

- Which instructions to use
- Which registers to use

Instructions as Numbers (1/2)

- Currently all data we work with is in words (32-bit blocks):
  - Each register is a word.
  - **lw** and **sw** both access memory one word at a time.
- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so “add $t0,$0,$0” is meaningless.
  - MIPS wants simplicity: since data is in words, make instructions be words...

Instructions as Numbers (2/2)

- One word is 32 bits, so divide instruction word into “fields”.
- Each field tells computer something about instruction.
- We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  - R-format
  - I-format
  - J-format

Internal representation

- **Elements of an addition**
  - **Source register 1**
  - **Source register 2**
  - **Target register**
  - **Operation type**
- **Example: add $8, $17, $18**

operation1	source1	source2	destination	unused	operation2
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

Operands

- Variables
  - Variables are mapped to registers
  - Registers can be accessed directly from the program
- Registers
  - Fixed, identical length (typical: 32 bits)
  - Fixed number (typical 32)
- MIPS
  - 32 registers with 32 bits
  - 30 usable, 2 with special meaning

Instructions: Groups

- **Arithmetic operations**
  - **Addition, Subtraction,...**
- **Information flow**
  - **Load from memory**
  - **Store in memory**
- **Logic operations**
  - **Logic and / or**
  - **Negation**
  - **Shift**
- **Branch operation**

Instructions: Types

- Instructions with different numbers of operands
  - 1 Operand:
    - Jump # address
    - Jump $ register number
  - 2 Operands:
    - Multiply $2, $3
    - Multiply $2 and $3 and store the result in ($2,$3)
  - 3 Operands:
    - Add a, b, c   # a = b + c
    - Add a, a, b   # a = a + b
    - Sub a, b ,c   # a = b - c

Assembly Variables: Registers (1/3)

- Unlike HLL, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Assembly Operands are registers
  - limited number of special locations built directly into the hardware
  - operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they are very fast

Assembly Variables: Registers (2/3)

- 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
- 32 registers in MIPS
  - Why 32? Smaller is faster
- Each MIPS register is 32 bits wide
  - Groups of 32 bits called a word in MIPS

Assembly Variables: Registers (3/3)

- Registers are numbered from 0 to 31
- Number references:
  - $0, $1, $2, … $30, $31
- By convention, each register also has a name to make it easier to code:
  - $16 - $23 ➔ $s0 - $s7
    (correspond to Java variables)
  - $8 - $15 ➔ $t0 - $t7
    (correspond to temporary variables)
- In general, use register names to make your code more readable

Assembly Design: Key Concepts

- Assembly language is essentially directly supported in hardware, therefore ...
- It is kept very simple!
  - Limit on the type of operands
  - Limit on the set operations that can be done to absolute minimum
    - if an operation can be decomposed into a simpler operation, don't include it

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 Java
  - The equivalent of # is //
  - Java comments have also the 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 Java (and most other High Level Languages), where each line could represent multiple operations

Content

- *Review*
- *Again on the Stored Program concept*
- *Program preparation and structure of the instructions*
- **Simple math operations**

Review of Java Operators/Operands

- Operators: +, -, *, /, % (mod);
  - 7/4==1, 7%4==3
- Operands:
  - Variables: fahr, celsius
  - Constants: 0, 1000, -17, 15.4
- Assignment Statement:
  - Variable = expression
  - Examples:
    - celsius = 5*(fahr-32)/9;
    - a = b+c+d-e;

Addition and Subtraction (1/3)

Syntax of Instructions:

- 1 2,3,4
  where:
  1) operation by name
  2) operand getting result ("destination")
  3) 1st operand for operation ("source1")
  4) 2nd operand for operation ("source2")

Syntax is **rigid**:

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

Addition and Subtraction (2/3)

- Addition in Assembly

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

  Equivalent to: a = b + c (in Java)

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

- Subtraction in Assembly

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

  Equivalent to: d = e - f (in Java)

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

Addition and Subtraction (3/3)

- How do the following C statement?
  a = b + c + d - e;
- Break into multiple instructions

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

Notice: A single line of Java may break up into several lines of MIPS.

Notice: Everything after the hash mark on each line is ignored (comments)

Example

- **Add the four variables b, c, d, and e and place the result in variable a.**

  a = b + c + d + e

- **Add operation, first parameter is the destination, second + third are the source**
- **Sequence:**
  - **add $s0, $s1, $s2   # a = b + c**
  - **add $s0, $s0, $s3   # a = b + c + d**
  - **add $s0, $s0, $s4   # a = b + c + d + e**

Proposed exercises

Transform the following Java instructions into assembly code

- a = a + b – c;
- a = (b-c)+(e-f)
- a--;
- b+=c;
- a = 2*(b-c)+d;

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 Java)

  where registers \$s0,\$s1 are associated with variables f, g
- Syntax similar to add instruction, except that last argument is a number instead of a register

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
- Use this register, it's very handy!
  - add $6,$0,$5 # copy $5 to $6
- This register is defined in hardware, so an instruction like
  - addi $0,$0,5

  will not do anything

Remember the different representations

- **Assembler code: Mnemonic notation**
  - **add $8, $18, $8**
  - **sub $9, $12, $15**
- **Binary representation**
  - **Sequence of 32 '1's and '0's**

    0111 0000 1001 0100 1010 1001 0100 1010

More proposed exercises

Transform the following Java instructions into assembly code

- a +=23;
- a = 2*(b+c) – 4 – d
- b = 27+15+d
- a = (4/3)*3.14*b*b*b