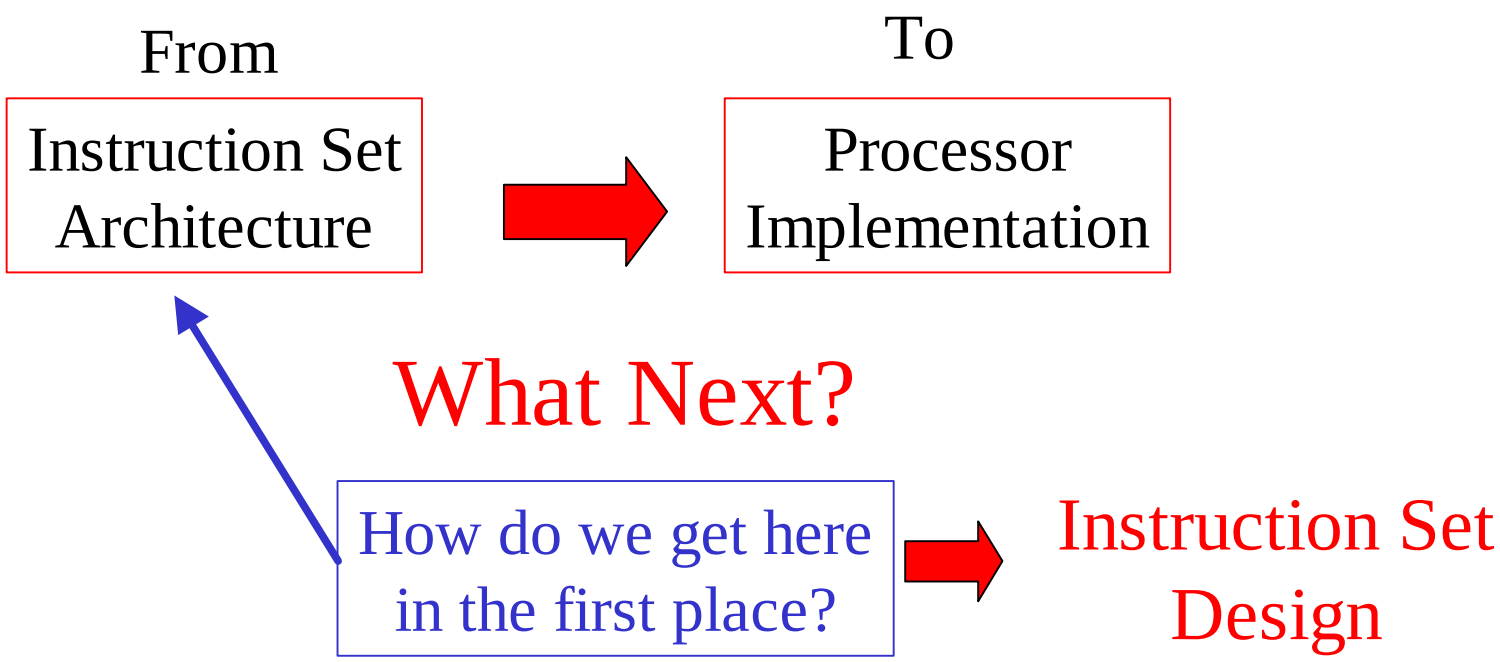
Programming Universal Computers
Instruction Sets

Lecture 5

Prof. Bienvenido Velez

What do we know?



Outline

- Virtual Machines: Interpretation Revisited
- Example: From HLL to Machine Code
- Implementing HLL Abstractions
  - Control structures
  - Data Structures
  - Procedures and Functions

Virtual Machines (VM s)

Type of Virtual Machine	Examples	Instruction Elements	Data Elements	Comments
Application Programs	Spreadsheet, Word Processor	Drag & Drop, GUI ops, macros	cells, paragraphs, sections	Visual, Graphical, Interactive Application Specific Abstractions Easy for Humans Hides HLL Level
High-Level Language	C, C++, Java, FORTRAN, Pascal	if-then-else, procedures, loops	arrays, structures	Modular, Structured, Model Human Language/Thought General Purpose Abstractions Hides Lower Levels
Assembly-Level	SPIM, MASM	directives, pseudo-instructions, macros	registers, labelled memory cells	Symbolic Instructions/Data Hides some machine details like alignment, address calculations Exposes Machine ISA
Machine-Level (ISA)	MIPS, Intel 80x86	load, store, add, branch	bits, binary addresses	Numeric, Binary Difficult for Humans

Computer Science in Perspective

People

Application Programs

High Level

INEL

4206

Language (ISA)

computers

Computer Human Interaction, User Interfaces

CS1/CS2, Programming, Data Structures

Programming Languages, Compilers

Computer Architecture

INTERPRETATION
A **CORE** theme all throughout Computer Science

Computing Integer Division
Iterative C++ Version

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

We ignore procedures and I/O for now

Easy I
A Simple Accumulator Processor
Instruction Set Architecture (ISA)

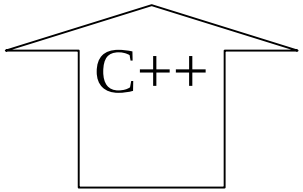
Instruction Set

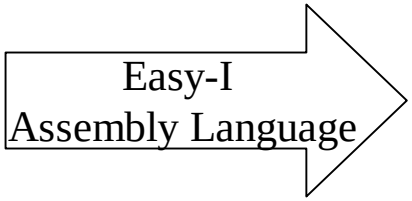
Symbolic Name	Opcode	Action I=0	Symbolic Name	Action I=1
Comp	00 000	AC ? not AC	Comp	AC <- not AC
ShR	00 001	AC ? AC / 2	ShR	AC ? AC / 2
BrN	00 010	AC < 0 ? PC ? X	BrN	AC < 0 ? PC ? MEM[X]
Jump	00 011	PC ? X	Jump	PC ? MEM[X]
Store	00 100	MEM[X] ? AC	Store	MEM[MEM[X]] ? AC
Load	00 101	AC ? MEM[X]	Load	AC ? MEM[MEM[X]]
Andc	00 110	AC ? AC and X	And	AC ? AC and MEM[X]
Addc	00 111	AC ? AC + X	Add	AC ? AC + MEM[X]

Computing Integer Division

Iterative C++ Version

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```





Computing Integer Division

Iterative C++ Version

Translate Data: Global Layout

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++

Easy-I
Assembly Language

```
0:      andc 0          # AC = 0
        addc 12
        store 1000      # a = 12 (a   tored @ 1000)
        andc 0          # AC = 0
        addc 4
        store 1004      # b = 12     stored @ 1004)
        andc 0          # AC =
        store 1008      # res        0 (result @ 1008)
```

Issues
- Memory allocation
- Data Alignment
- Data Sizing

Computing Integer Division
Iterative C++ Version

Translate Code: Conditionals If-Then

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++

Easy-I Assembly Language

```
0:      andc 0          # AC = 0
        addc 12
        store 1000      # a = 12 (a stored @ 1000)
        andc 0          # AC = 0
        addc 4
        store 1004      # b = 12 (b stored @ 1004)
        andc 0          # AC = 0
        store 1008      # result = 0 (result @ 1008)
main:   load 1004       # compute a – b in AC
        comp            # using 2's complement add
        addc 1
        add 1004
        brn exit        # exit if AC negative




exit:
```

Issues
- Must translate HLL boolean expression into ISA-level branching condition

Computing Integer Division
Iterative C++ Version

Translate Code: Iteration (loops)

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++

Easy-I
Assembly Language

```
0:      andc 0          # AC = 0
        addc 12
        store 1000      # a = 12 (a stored @ 1000)
        andc 0          # AC = 0
        addc 4
        store 1004      # b = 12 (b stored @ 1004)
        andc 0          # AC = 0
        store 1008      # result = 0 (result @ 1008)
main:   load 1004       # compute a – b in AC
        comp            # using 2's complement add
        addc 1
        add 1004
        brn exit        # exit if AC negative
loop:   load 1000
        brn endloop




        jump loop
endloop:
exit:
```

Computing Integer Division
Iterative C++ Version

Translate Code: Arithmetic Ops

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++

Easy-I
Assembly Language

```
0:      andc 0              # AC = 0
        addc 12
        store 1000          # a = 12 (a stored @ 1000)
        andc 0              # AC = 0
        addc 4
        store 1004          # b = 12 (b stored @ 1004)
        andc 0              # AC = 0
        store 1008          # result = 0 (result @ 1008)
main:   load 1004           # compute a – b in AC
        comp                # using 2's complement add
        addc 1
        add 1004
        brn exit            # exit if AC negative
loop:   load 1000
        brn endloop
        load 1004
        comp
        add 1004            # Uses indirect bit I = 1




        jump loop
endloop:
exit:
```

Computing Integer Division
Iterative C++ Version

Translate Code: Assignments

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++

Easy-I
Assembly Language

```
0:      andc 0              # AC = 0
        addc 12
        store 1000          # a = 12 (a stored @ 1000)
        andc 0              # AC = 0
        addc 4
        store 1004          # b = 12 (b stored @ 1004)
        andc 0              # AC = 0
        store 1008          # result = 0 (result @ 1008)
main:   load 1004           # compute a – b in AC
        comp                # using 2's complement add
        addc 1
        add 1004
        brn exit            # exit if AC negative
loop:   load 1000
        brn endloop
        load 1004
        comp
        add 1004            # Uses indirect bit I = 1
        store 1000




        jump loop
endloop:
exit:
```

Computing Integer Division
Iterative C++ Version

Translate Code: Increments

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++

Easy-I
Assembly Language

```
0:      andc 0          # AC = 0
        addc 12
        store 1000      # a = 12 (a stored @ 1000)
        andc 0          # AC = 0
        addc 4
        store 1004      # b = 12 (b stored @ 1004)
        andc 0          # AC = 0
        store 1008      # result = 0 (result @ 1008)
main:   load 1004       # compute a – b in AC
        comp            # using 2's complement add
        addc 1
        add 1004
        brn exit        # exit if AC negative
loop:   load 1000
        brn endloop
        load 1004
        comp
        add 1004        # Uses indirect bit I = 1
        store 1000
        load 1012       # result = result + 1
        addc 1
        store 1012
        jump loop
endloop:
exit:
```

Computing Integer Division

Easy I Machine Code

Data

Address	Contents
1000	a
1004	b
1008	result

Challenge

Make this program as small and fast as possible

Address	I Bit	Opcode (binary)	X (base 10)
0	0	00 110	0
2	0	00 111	12
4	0	00 100	1000
6	0	00 110	0
8	0	00 111	4
10	0	00 100	1004
12	0	00 110	0
14	0	00 100	1008
16	0	00 101	1004
18	0	00 000	unused
20	0	00 111	1
22	1	00 111	1004
24	0	00 010	44
26	0	00 101	1000
28	0	00 010	44
30	0	00 101	1004
32	0	00 000	unused
34	1	00 111	1004
36	0	00 100	1008
38	0	00 101	1012
40	0	00 111	1
42	0	00 101	1000
44	0	00 011	26

Program

The MIPS Architecture

- Reduced Instruction Set Computer (RISC)
- 32 general purpose registers
- Load-store architecture: Operands in registers
- Simple and Uniform instruction formats
  - R Format: Arithmetic/Logic operations on registers

  - I Format: Branches, loads and stores

SPIM Assembler

- Symbolic Labels
- Assembler Directives
- Pseudo-Instructions
- Macros

Computing Integer Division
Iterative C++ Version

Step 1: Layout Global Data

C++

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

SPIM Assembler

```
          .data
a         .word    12
b         .word    4
result    .word    0
          .glob    main     #global symbol
```

Computing Integer Division
Iterative C++ Version

Step 2: Translate If-Then-Else

C++

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

SPIM Assembler

```
          .data               # data segment
a:        .word    12
b:        .word    4
result:   .word    0
          .glob    main       # global symbols

          .text               # text segment
main:
```