Computer Systems
CSE 410 Winter 2022

RISC-V Instruction Set Architecture (ISA)
Today’s Lecture Guide

I. RISC-V Instruction Set Architecture (ISA): Resources

II. RISC-V ISA: Instructions

III. The Assembler

IV. The CSE 410 Simulator
I. RISC-V ISA: Resources

Instruction Set Architecture (ISA)

software

hardware
Instruction Set Architecture (ISA)

- An ISA is an abstract interface
  - It’s not an implementation
  - Allows executables to be portable across processor models

- The ISA defines the set of resources available, the set of operations available, and what you have to do to make use of those resources and operations

- We’ll be talking about RISC-V, but other architectures look very similar, at least for the purposes of this class
  - Roughly same resources
  - Roughly same operations
The Context

- There’s the full RISC-V ISA
  - It’s defined in the spec (risc-v.org), but has way more in it than is needed in this course
  - RISC-V itself isn’t one ISA, it’s a family
    - For instance, there are 32-bit ISA specs, and 64-bit and 128-bit (What does that mean?)

- There’s the subset of the RISC-V ISA that we will eventually cover
  - It is basically the user-level (not instructions intended for the OS to use), integer (not float) instructions
  - We’re leaving out instructions even from that subset to help simplify
    - Nothing essential is missing, though
  - Our subset is from RV32
The Context (cont.)

- We’ll be using a set of recently written tools for programming on the RISC-V ISA

- Why?
  - The tools simplify the architecture even more than just subsetting RV32
    - Basically, they let you program while thinking in decimal
      - (“Real programmers use hexadecimal”)
      - (We’ll get to that in a while...)

- So, the way our simulated machine works in hw1 differs from what you’ll read about RISC-V online

- The goal of the simplification is that you shouldn’t have to read (much) beyond the hw writeup for hw1

- For the rest of these slides, when I say RISC-V I mean the version implemented in our tools for hw1
RISC-V Resources

CPU
(Central Processing Unit)

Registers

PC
(program counter)

Memory bus

(Main) Memory
(RAM (random access memory))
Hardware Packaging: CPU
Hardware Packaging: CPU
Hardware Packaging: CPU

1978 VS 2018

- 29,000 Transistors
- 3 Micron NMOS Process (3000 NM)
- 6-Line (Mercury ABC Lamps), 436NM Wavelength
- 33 MM²
- 3.2 Microns (3200 NM)
- 5 MHz (0.005 GHz)
- 0.33 MIPS
- 4 INCHES
- $86 ($330 Adjusted for Inflation)

- BILLIONS
- 14 NM CMOS Process
- Argon Fluoride Excimer Laser, 193NM Wavelength
- > 100 MM²
- 8 NM
- 5 GHz
- > 100,000 MIPS
- 12 INCHES
- $425 MSRP

https://www.legitreviews.com/intel-core-i7-8086k-processor-review_206547
Hardware Packaging: RAM
Hardware Packaging: Motherboard

- Memory slots: quad channel DDR4 4400 (O.C.), 4266 (O.C.), 4133 (O.C.), 4000 (O.C.), 3866 (O.C.), 3733 (O.C.), 3600 (O.C.), 3466 (O.C.), 3400 (O.C.), 3333 (O.C.), 3300 (O.C.), 3200/2933/2667/2400/2133
- CPU socket: sTRX4
- 8-pin CPU power connectors
- Memory slots: quad channel DDR4 4400 (O.C.), 4266 (O.C.), 4133 (O.C.), 4000 (O.C.), 3866 (O.C.), 3733 (O.C.), 3600 (O.C.), 3466 (O.C.), 3400 (O.C.), 3333 (O.C.), 3300 (O.C.), 3200/2933/2667/2400/2133
- 24-pin ATX power connector
- Chipset: AMD TRX40
- SATA connectors: 6Gb/s
- M.2 slot
- PCIe Express 4.0 x1 slot
- M.2 slot
- PCIe Express 4.0 x16 slots
RISC-V (RV32i) ISA

- Each register can hold a Java integer

- Each unit of memory can hold a Java integer
  - We call a unit of memory a “word”

- You know from 142/143 that there is a limited range of integers that a Java int can hold
  - -2,147,483,648 to 2,147,483,647

- Why?
RISC-V (RV32i) Resources

CPU
(Central Processing Unit)

Registers

PC
(program counter)

Memory bus

(Main) Memory
(RAM (random access memory))

Each register/word of memory is 32-bits wide

01010101010101010101010101010101
Binary integer preview

- Why is the range of integers a 32-bit int can hold limited?
  - Let’s answer that using 3-bit integers as an example

- If each register/word were 3 bits wide, each could hold any of 8 different bit patterns
  - 000, 001, 010, 011, 100, 101, 110, 111

- So, the register/word can hold only 8 different integer values
  - What values should they be?
Possible bit string to integer mappings

<table>
<thead>
<tr>
<th>Bit String</th>
<th>Option A</th>
<th>Option B</th>
<th>Option C</th>
<th>Option D</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>10</td>
<td>-17</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>100</td>
<td>6,513,201</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>011</td>
<td>1,000</td>
<td>8</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>10,000</td>
<td>-2</td>
<td>4</td>
<td>-4</td>
</tr>
<tr>
<td>101</td>
<td>100,000</td>
<td>12</td>
<td>5</td>
<td>-3</td>
</tr>
<tr>
<td>110</td>
<td>1,000,000</td>
<td>10</td>
<td>6</td>
<td>-2</td>
</tr>
<tr>
<td>111</td>
<td>10,000,000</td>
<td>6</td>
<td>7</td>
<td>-1</td>
</tr>
</tbody>
</table>

Option D is called “two’s complement representation” and corresponds to the Java int’s you’re used to

Option C is called “unsigned int”

Option A is related to floats. Option B is silly. (Why?)
(Sim) RISC-V Resources: Naming

There are actually 32 registers. The amount of memory is system dependent.

CPU
(Central Processing Unit)

Registers

PC
(program counter)

Memory bus

(Main) Memory
(RAM (random access memory))

0
1
2
3
4
5
6
7
8
9
10
11
12
13
(Sim) RISC-V Resources: Naming

Example Instruction:
“Add register 3 to register 4 and put the result in register 3”
Example Instruction:
“Copy the bits in memory location 10 to register 7”
(not a real instruction)
(Sim) RISC-V Resources: **Addressing**

**Example Instruction:**

"Copy the bits in the memory location whose address is in register 3 to register 7"
Pointers

Registers are used to name words of memory. Registers can hold only about 4B different names => The processor can’t use memory bigger than about 4 giga-words
RISC-V Resources Summary

- **32 registers**
  - Named by the hardware as 0, 1, 2, ..., 31
  - **Register 0 is always 0**
    - if you read its value, you get 0
    - if you write any value to it, it stays 0
  - Register contents can be used as pointers to name memory locations

- **Some number of words of memory**

- **A program counter (PC)**
  - The PC is a pointer to (names) a memory location
  - That location is the next instruction to be executed
II. RISC-V ISA: Instructions
Everything is a Bit String

- Registers and memory hold bit strings
- The hardware manipulates bit strings
  - At run time, everything is bits
- Layers above map bit strings to other sorts of data
  - *Layers:*
    - Some software package
    - The compiler
    - The assembler
    - The programmer
  - *Other sorts of data:*
    - Colors (Red: 00, Blue: 01, Green: 10)
    - Animals (Dog: 00, Cat: 01, Rat: 10, Squirrel: 11)
    - Boolean (False: 0, True: 1)
An Executable’s Instructions are Bit Strings

- When a program runs, its instructions are stored in memory
- Example:
  - 00000000111101110000011110110011 means
    - *add the contents of registers 14 and 15 and put the result in register 15*

- The hardware directly executes these bit strings, which are called *machine instructions*

- Machine instructions have
  - An op(eration) code – e.g., add
  - Operands – e.g., registers 3 and 4

- The special program counter (PC) register in the CPU is a pointer to (holds the address of) the next instruction to execute
Operation of the CPU / Control Flow

- Basic operation of the CPU: fetch-increment-execute
  - Fetch the instruction memory pointed to by the program counter (PC)
  - PC ← PC + 1
  - Execute the instruction just fetched
  - Repeat (forever)

- Note that instructions are usually executed sequentially
  - Instruction at location 113, 114, 115, 116, ...

- A “branch instruction” is a (conditional) assignment to the PC
  - Executing instructions at location 113, 114, 210, 211, ...
    - The instruction at 114 was a branch
  - (Terminology: “branch” is conditional on some test; “jump” is unconditional)
Simple Example Assembly Program

.text

...  
lw x1, X  
lw x2, Y 
add x2, x1, x2
bge x2, x0, else
...//then
j done
else: ...// else
done: sw x2, Z

.data
X: .word 1
Y: .word 2
Z: .word 0

int X=1;
int Y=2;
...
it temp = X+Y;
if ( temp < 0) {
    //then
}
else {
    //isneg
}

Z = temp;
RISC-V Instructions

- **RISC** is Reduced Instruction Set Computer
  - It’s contrasted with CISC, Complex Instruction Set Computer

- RISC is designed to allow extremely fast implementations of the ISA
  - Fast comes from regularity and simplicity
    - “Simpler is faster”

- Instruction types:
  - A) Memory operations: move values between registers and memory
  - B) Reg-Reg operations: perform an operation on register values and store result in a register
  - C) Control Flow (branches): compare two registers and if result is True, assign a value to PC
A) Memory Operations: Summary

- **Load** moves data from memory into a register
- **Store** moves data from a register into memory

**Assembler Format:**

- `lw rd, imm(rs1)  # rd ← memory[imm+[rs1]]`
- `sw rs2, imm(rs1)  # memory[imm+[rs1]] ← [rs2]`
**Memory Operations: Load**

Example: load word

```assembly
lw x5, 4(x3)  # load into register 5 the word in
              # memory at location given by
              # adding 4 to the contents of reg 3
```
Base-Displacement Addressing Mode

lw x5, 4(x3)  # load into register 5 the word in memory at location
   # given by adding 4 to the contents of reg 3

- An “addressing mode” is a scheme to create a memory address
- Base-displacement adds a constant (given in the instruction) to
  the contents of a register
- The register is called “the base register” because typically it
  points to the base of some data structure
- Example: x3 holds the address of the zero-th word of array A.
  Then 4(x3) names A[4].
Memory Operations: Store

sw x5, 3(x3)  # store the contents of reg 5
# memory at location given by
# adding 3 to the contents of reg 3
B) Register Operations: Summary

- **op rd, rs1, rs2**
  - \([rd] \leftarrow [rs1] \text{ op } [rs2]\)
    - The contents of register \(rd\) are replaced with the value obtained by performing the operation on the contents of registers \(rs1\) and \(rs2\)

- **Example arithmetic instructions**
  - `add x3, x1, x2`  # \(x3 = [x1] + [x2]\)
  - `sub x3, x1, x2`  # \(x3 = [x1] - [x2]\)
  - `and x3, x1, x2`  # \(x3 = [x1] \& [x2]\)
  - `or x3, x1, x2`  # \(x3 = [x1] \mid [x2]\)
  - `xor x3, x1, x2`  # \(x3 = [x1] \^ [x2]\)
  - `sll x3, x1, x2`  # \(x3 = [x1] << [x2]\)
  - `sra x3, x1, x2`  # \(x3 = [x1] >> [x2]\)

...
Immediate Instructions: Summary

- An “immediate” is a (small) value stored in the instruction itself
  - Remember, instructions are bit strings stored in memory
  - “Big values” are too big to store in the instruction, and so must be put in memory and loaded into registers when you want to use them

- \( \text{op} \quad \text{rd}, \text{rs}1, \text{immed} \)
  - \([\text{rd}] \leftarrow [\text{rs}1] \text{ op immed} \)
    - The contents of register \( \text{rd} \) are replaced with the value obtained by performing the operation on the contents of registers \( \text{rs}1 \) and the immediate value
Immediate Instructions: Examples

Example immediate instructions

- addi x3, x1, 1  # x3 = x1 + 1
  addi x3, x1, -1  # x3 = x1 - 1
- andi x3, x1, 1  # x3 = x1 & 1
- ori  x3, x1, 1  # x3 = x1 | 1
- srai x3, x1, 1  # x3 = x1 >> 1
- ...

...
C) Control Flow (branch instructions): Summary

- `blt x1, x2, 20`  # if [x1] < [x2], PC ← 20
- `bge x1, x2, 10`  # if [x1] > [x2], PC ← 10
- `beq x1, x2, 21002`  # branch if [x1] == [x2]
- `bne x1, x2, 104`  # branch if [x1] != [x2]

- **Reminder:** [x0] == 0
  It’s always 0, even if some program tries to assign to it.

- **The assembler supports some “pseudo-instructions”**
  - `beqz x1, 20`  # branch if [x1] == 0
  - `beq x1, x0, 20`  # branch if [x1] == 0
Control Flow (branch instructions)

- `blt  x1, x2, 20`  # if [x1] < [x2], PC ← 20
- `bge  x1, x2, loop`  # if [x1] > [x2], PC gets the location with label loop
- `beq  x1, x2, done`
- `bne  x1, x2, start`

`done:`  # example of a label

*Note: These are instructions written in a format the assembler understands. To the assembler, a label is basically a symbolic constant whose value is the memory location the label names. So, if `loop` names memory location 12, the second instruction is equivalent to `bge  x1, x2, 12`*
III. The Assembler
An Executable’s Instructions are Bit Strings

- **Assembler**: generally translates a single assembly language instruction (written as characters) into a single machine instruction (a bit string).
- **Compiler**: generally translates source language instructions into sequences of assembly language instructions and does type checking.
Assembly Language Instructions

- Format: [label:] opcode operands
- Examples:
  
  ```
  lw x9, 0(x3)
  beqz x9, skip
  addi x9, x9, -1
  sw x9, 0(x3)
  skip: ...
  ```
The Assembler: Labels

- Labels are assembler-supported symbolic names for instruction addresses

  \[
  \text{lw} \quad x9, 0(x3) \\
  \text{beqz} \quad x9, \text{skip} \\
  \text{addi} \quad x9, x9, -1 \\
  \text{sw} \quad x9, 0(x3)
  \]

  \text{skip:} \quad ... 

- If these instructions are stored starting at address 0, the value of the symbol “skip” is 4
  - Relieves programmer of burden of calculating addresses when writing branches
  - Assembler program remains correct even if a new instruction is inserted into the sequence
    - (A new instruction changes the addresses of all following instructions)
Assembler Pseudo-Instructions

The assembler allows the user to write instructions that aren’t directly supported by the CPU, because they can be easily expressed as instructions that are supported.

- `bgt x1, x2, loop` is just `blt x2, x1, loop`
- `ble x1, x2, loop` is `bge x2, x1, loop`
- `mov x1, x2` is `addi x1, x2, 0`
- `nop (no op(eration))` is `addi x1, x0, 0`
- `neg x1, x2` is `sub x1, x0, x2`
- `beqz x1, address` is `beq x1, x0, address`
- `j (jump) loop` is `beq x0, x0, loop`

(Note: this isn’t the translation actually used for j, but it’s correct)
Example Assembler Program: hw0.asm

```
.text
main:
    lw x5, x
    lw x6, y
    add x5, x5, x6
    lw x6, z
    add x5, x5, x6
    print x5
    halt # all done

.data
x: .word 1
y: .word 2
z: .word 3
```

section directives
labels
comments
Assembler File Sections

- Overview
  - `.text` means that what follows are instructions intended to be executed, not data
  - `.data` means what follows is data, not instructions

- `.data` Section
  - `.word 3` means that a word of memory should be initialized (before the program starts running) with the value 3
Labels for Data

- Remember that load and store operations need to compute the location in memory that will be operated on
  - E.g., 4(x3) means $4 + [x3]$
  - The computed location is called “the effective address”
  - This style of addressing is called “base displacement”

- It is unwieldy for humans (especially) to deal with addresses
  - If you insert or delete a variable, they all change

- Labels are a convenience, provided by the assembler
  - The assembler keeps track of the lines of code and words of data it has seen so far, and when it sees a label it remembers the location that was labelled
Labels

Label “main” means location 0 in the text segment

Labels x, y, and z means locations 0, 1, and 2 in the data segment
IV. The CSE 410 Simulator
Why Our Own Simulator?

- There are many RISC-V tool kits
- There are even RISC-V processors

Our simulator is a simplified RISC-V ISA:
  - The essential functionality of RISC-V RV32i without a lot of distracting details
  - The essential RISC ideas without...

Makes it possible to actually do a small implementation without requiring weeks of instruction
  - I hope!
Instructions Supported by the HW1 Simulator

- RISC-V RV32 Instructions
  - add, sub, addi, lw, sw, blt, bge, beq, bne

- Made up instructions
  For simplicity, the CSE 410 simulator invents a couple instructions that real processors don’t have
  - halt # stop execution
  - print x5 # print [x5] to the console
Example program: hw0.asm

```
.text
main:
    lw     x5, x
    lw     x6, y
    add    x5, x5, x6
    lw     x6, z
    add    x5, x5, x6
    print  x5
    halt

.data
x:     .word  1
y:     .word  2
z:     .word  3
```

*In this example, execution starts at location 0 in the text section*

*In the future, execution of your code will start at label main*

*Program termination*
Running the Simulator

$ Sim hw0.asm

Runs hw0.asm. When execution halts, the CPU state (PC and register values) and memory state (contents) are printed. Values of 0 are largely elided.

$ Sim hw0.asm
6
Cycle: 7
7 instructions executed
PC = 7
[reg:x0] 0
...
[reg:x2] 524288
[reg:x3] 0
...
[reg:x5] 6
[reg:x6] 3
[reg:x7] 0
...
[reg:x31] 0
[mem:0] 1
[mem:1] 2
[mem:2] 3
Running the sim debugger

$ Sim -d hw0.asm
Welcome to the Sim410 debugger. Type help or ? to list commands.

(pc: 0 cycle: 0) ?

Documented commands (type help <topic>):
========================================EOF
dcpu dinstructions dmemory dregs help run step

(pc: 0 cycle: 0) help step
Execute one instruction
(pc: 0 cycle: 0) step
[hw0.asm: 2] lw x5, x
(pc: 1 cycle: 1) dregs
[reg:x0] 0
... [reg:x2] 524288
[reg:x3] 0
... [reg:x5] 1
[reg:x6] 0
... [reg:x31] 0
(pc: 1 cycle: 1)

Type ctrl-d to exit
Just run, but show instructions executed

$ Sim -t hw0.asm

[hw0.asm: 2] lw x5, x
[hw0.asm: 3] lw x6, y
[hw0.asm: 4] add x5, x5, x6
[hw0.asm: 5] lw x6, z
[hw0.asm: 6] add x5, x5, x6
[hw0.asm: 7] print x5

6
[hw0.asm: 8] halt

Cycle: 7
7 instructions executed
PC = 7
[reg:x0] 0
...
[reg:x2] 524288
[reg:x3] 0
...
[reg:x5] 6
[reg:x6] 3
[reg:x7] 0
...
[reg:x31] 0
[mem:0] 1
[mem:1] 2
[mem:2] 3
Another example assembler program

.text
    addi  x5, x0, 10
loop:   print  x5
    addi  x5, x5, -1
    bne   x5, x0, loop
halt

*What does execution of this program print?*
*What is the value of the symbol ‘loop’?*
*Where does execution start?*
Final Example

.text
addi  x5, x0, head
addi  x6, x0, tail
sub   x5, x5, x6
loop: print x5
      halt
.data
head: .word 0
      .word 10
      .word -3
tail:  .word 9

What does execution of this program print?
Warning: for hw1 you can completely ignore what’s said here.

There are two big, one medium, and one small simplifications made by the simulator at this point:

1. Memory is not addressed by word, but by byte. A byte is 8 bits, ¼ of a word. So, address 1 isn’t the second word of memory, it’s the second byte. The second word of memory is in bytes 4 through 7. *We’re simulating word addressable memory now because it’s easier for humans to count by 1 than by 4 while debugging.*

2. In real systems, instructions and data share the same memory. So, the first word in the data section is not at location 0 in memory. In the simulator, the data and text segments are distinct, and each starts at its own location 0. *Again, this makes debugging easier because you don’t have to count over the instructions to figure out the locations of the data words.*

3. The simulator doesn’t actually put instructions in memory. It executes the assembly instructions directly. It’ll be clear why in a week or two.

4. We’re kind of lying about how branch target addresses are computed, but in a way that is irrelevant now and that I’ll be able to justify once I tell you how they actually work.
Big things we’ve left out that are in RISC-V

- Multiply/divide and the like
  - Adding that to the ISA seems to result in instructions of varying lengths. We want all instructions to be the same length.

- Floating point operations
  - RISC-V specifies float operations and float registers. We’ll just live without them.

- Operations required to implement the operating system

- Atomic instructions / Multicore support
  - Thread safe operations needed for multicore processors