Computer Systems
CSE 410 Winter 2020

RISC-V ISA
Instruction Set Architecture

- An ISA is an abstract interface
  - It’s not an implementation
  - But we often talk about it as if it were

- 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 useful to this course

- There’s the subset of the RISC-V ISA that we will eventually cover
  - It is basically the user-level, integer instructions
  - We’re leaving out instructions even from that subset to help simplify
    - Nothing essential is missing, though
  - 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 (in progress)
  - 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)

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

Memory bus
Hardware Packaging: CPU
Hardware Packaging: CPU
Hardware Packaging: CPU

1978 vs 2018

- 29,000 Transistors
- Manufacturing
- Lithography
- Die Size
- Min Feature Size
- Max Frequency
- Performance
- Wafer Diameter
- Price

- 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

PCI Express 4.0 x16 slots

PCI Express 4.0 x1 slot

M.2 slot
RISC-V 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
HW0 RISC-V 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?
Binary integer preview

- Why is the range of integers a 32-bit in 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?
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 corresponds to the Java int's you're used to

Option C is called an “unsigned int”

Options A and B are silly. (Why?)
There are 32 registers. The amount of memory is system dependent.
RISC-V Resources

We'll see that 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 assembler) x0, x1, x2, ..., x31
  - x0 is always 0
    - if you read its value, you get 0
    - if you write any value to it, it stays 0

- **Some number of words of memory**

- **A program counter (PC)**
  - The PC names a memory location
  - That location is the next instruction to be executed
Operation of the CPU

- Basic operation of the CPU: fetch-increment-execute
  - Fetch the instruction located in memory at the location named by the 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
  - Instruction at location 113, 114, 210, 211, ...
    - The instruction at 114 was a branch
    - (Terminology: “branch” is conditional on some test; “jump” is an unconditional change in control flow)
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

- Instruction types:
  - Memory operations: move values between registers and memory
  - Reg-Reg operations: perform an operation on register values and store result in a register
  - Branches: compare two registers and if result is True, assign a value to PC
Memory Operations: Load

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

- op rd, rs1, rs2
  - [rd] ← [rs1] 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

- Arithmetic instructions
  - add x3, x1, x2  # x3 = x1 + x2
  - sub x3, x1, x2

- Immediate instructions
  - addi x3, x1, 1  # x3 = x1 + 1
  - addi x3, x1, -1
Branch instructions

- `blt x1, x2, 20` # if \([x1] < [x2]\), PC \(\leftarrow 20\)
- `bge x1, x2, loop` # if \([x1] > [x2]\), PC gets the location with # label loop
- `beq x1, x2, done`
- `bne x1, x2, start`
Other instructions

- **Made up instructions**
  For simplicity, the simulator invents a couple instructions that real processors don’t have
  
  - halt # stop execution
  - print x5 # print [x5] to the console

- **Additional real instructions**
  There are more actual RISC-V instructions implemented in the simulator. We’ll introduce them as we need them.
Unneeded instructions

- bgt \( x_1, x_2, \text{loop} \) is just blt \( x_2, x_1, \text{loop} \)
- ble \( x_1, x_2, \text{loop} \) is bge \( x_2, x_1, \text{loop} \)
- mov \( x_1, x_2 \) is just addi \( x_1, x_2, 0 \)
- nop is just addi \( x_1, x_0, 0 \)
- neg \( x_1, x_2 \) is sub \( x_1, x_0, x_2 \)
- beqz \( x_1, \text{offset} \) is beq \( x_1, x_0, \text{offset} \)
- j \( \text{loop} \) is beq \( x_0, x_0, \text{loop} \)
  (Note: this isn’t the translation actually used for j, but it works)
Example Assembler Program: hw0.asm

```assembly
.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

comment
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
  - This should seem magical at this point in the class. How is it initialized? What hardware or software component does the initialization? For now, let’s just accept some magic.
Labels

- 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”

- 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

```
Labels x, y, and z means locations 0, 1, and 2 in the data segment
```

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

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

```
Label “main” means location 0 in the text segment
```

```
Labels x, y, and z means locations 0, 1, and 2 in the data segment
```
$ Sim hw0.asm

Runs hw0.asm. Any print statements encountered during execution cause a value to be printed, but otherwise there is no output until the program halts. When it halts, it prints the CPU state (PC and register values) and memory state (contents). Values of 0 are largely elided.
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, \( \frac{1}{4} \) 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