CSCI 341 Computer Organization
– *Chapter 2: RISC-V Instruction Set Architecture*

I speak Spanish to God, Italian to women, French to men, and German to my horse.

Charles V, King of France

Topics for Chapter 2

- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS (Canvas resources)
  - Memory access instructions (Chapters 2.3)
  - Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 2:**
  - Procedure Calls (Chapters 2.8)
- **Part 3:**
  - RISC-V Instruction Format (Chapter 2.5, RISC-V reference card)
  - RISC-V Addressing Modes (Chapter 2.10)

Assembly Language

- **Assembly language vs. higher-level language**
  - Few, simple types of data and control
  - Does not specify variable type
  - Control flow is implemented with **goto/jump**
  - Assembly language programming
    - more difficult and error-prone,
    - machine-specific
    - longer

- **Assembly language vs. machine language**
  - Symbolic representation

- **When is assembly language needed**
  - Speed and size, (eg. Embedded computer)
  - Time-critical parts of a program
  - Specialized instructions

Instructions

- **Primitive operations that the CPU may execute**
- **Different CPUs implement different sets of instructions.**
- ***Instruction Set Architecture* (*ISA*):**
  - The set of instructions a particular CPU implements
  - Examples:
    - Intel 80x86 (Pentium 4)
    - IBM/Motorola PowerPC (Macintosh)
    - RISC-V
    - Intel IA64
    - ...

Instruction Set Architectures

- **CISC: Complex Instruction Set Computer (eg. 80x86)**
  - Instructions are of variable length and may be very complex
  - Instructions typically include string processing operations, complex record operations, etc.
  - Designed to ease of assembly language programming
  - Many memory-to-register/register-to-register operations
- **RISC: Reduced Instruction Set Computer (e.g., RISC-V, ARM, RISC-V)**
  - Cocke IBM, Patterson, Hennessy, 1980s
  - Keep the instruction set small and simple, making it easier to build faster hardware.
  - Let software do complicated operations by composing simpler ones.
  - Our focus: Open source RISC-V (https://riscv.org/)

Assembly Operands: Registers

- **Unlike HLL such as C or Java, assembly cannot use variables**
  - Keep Hardware Simple
- **Assembly Operands are registers**
  - Limited number of storage cells
    - RISC-V code must efficiently use registers
  - Built directly inside the processor
    - Directly accessible through machine instructions
    - Faster than memory

RISC-V Registers

- **32 registers**
  - Why 32? Smaller is faster
  - Can be referred to by number or name
    - Number references: `x0, x1, x2, … x30, x31`
    - Name references: make code more readable

      `x18 - x27` → `s0 - s11`

      (correspond to C variables, s for “saved”)

      `x28 - x31` → `t3 - t6`

      (correspond to intermediate results, t for “temporary”)
- **Each RISC-V register is 32 bits wide**
  - Groups of 32 bits called a word

RISC-V Registers

REGISTER	NAME	USE
x0	zero	Constant value 0
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5 - x7	t0 - t2	Temporaries
x8	s0/fp	Saved register/Frame pointer
x9	s1	Saved register
x10 - x11	a0 - a1	Function arguments/return values
x12 - x17	a2 - a7	Function arguments
x18 - x27	s2 - s11	Saved registers
x28 - x31	t3 - t6	Temporaries

Use names for registers -- code is clearer!

C 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

- **Hash (#) is used for RISC-V 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

Topics for Chapter 2

- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - ➔ Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS
  - Memory access instructions (Chapters 2.3)
  - Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 2:**
  - Procedure Calls (Chapters 2.8)
- **Part 3:**
  - RISC-V Instruction Format (Chapter 2.5)
  - RISC-V Addressing Modes (Chapter 2.10)

RISC-V Addition and Subtraction

- **Syntax of Instructions:**

  12,3,4

  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

RISC-V Addition and Subtraction

Addition in Assembly

- Example: `add s0,s1,s2` (in RISC-V)

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

  where RISC-V registers `s0,s1,s2` are associated with C variables `a, b, c`

Subtraction in Assembly

- Example: `sub s3,s4,s5` (in RISC-V)

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

  where RISC-V registers `s3,s4,s5` are associated with C variables `d, e, f`

RISC-V Addition and Subtraction: Example 1

- **How to 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 RISC-V.
  - Everything after the hash mark on each line is ignored (comments)

RISC-V Addition and Subtraction: Example 2

- **How to 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

- **Register zero (`x0` or `zero`) always has the value 0**
- **Example:**

  `add s0,s1,zero` (in RISC-V)

  `f = g` (in C)

  where RISC-V registers `s0,s1` are associated with C variables `f, g`

- **Defined in hardware, so an instruction**

  `add zero,zero,s0`

  will not do anything!

Immediates

- **Numerical constants.**
- **Appear often in code, so there are special instructions for them.**
- **Add Immediate:**

  `addi s0,s1,10` (in RISC-V)

  `f = g + 10` (in C)

  where RISC-V 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.**
- **Limited to 12-bit binary values**

Immediates

- **No Subtract Immediate in RISC-V**
  - 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 RISC-V)**

  `f = g - 10` (in C)

  where RISC-V registers `s0,s1` are associated with C variables `f, g`

RISC-V Reference Card

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	

Topics for Chapter 2

- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - ➔ Introduction to RARS (Canvas)
  - Memory access instructions (Chapters 2.3)
  - Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 2:**
  - Procedure Calls (Chapters 2.8)
- **Part 3:**
  - RISC-V Instruction Format (Chapter 2.5)
  - RISC-V Addressing Modes (Chapter 2.10)

RISC-V Simulators

- **Software simulator for processors that implement RISC-V architecture**
  - Read and execute assembly language files
- **RARS (we will use)**
  - a lightweight IDE for programming in RISC-V assembly language
  - Check Canvas for RARS setup instructions
- **Highlights**
  - Assembler directives (pseudo operations)
  - System calls
  - Pseudo instructions and enhanced instructions
    - Turn this off in settings menu to check your work
    - Not all pseudo instructions are allowed in this class (la unavoidable)

Assembler Syntax

- Program includes .data and .text
- Identifier names are sequence of letters, numbers, underbars (_) and dots (.).
- Labels are declared by putting them at beginning of line followed by a colon. Use labels for variables and code locations.
- Operands may be literal values or registers.
- Register is hardware primitive, can stored 32-bit value: s0
- Numbers are base 10 by default. 0x prefix indicates hexadecimal.
- Strings are enclosed in quotes. May include \n=newline or \t=tab. Used for prompts.

Assembler Directives

- .data <addr>
- .text <addr>
- .globl sym
- .ascii “str”: produces string NOT null terminated
- .asciz “str”: produces null-terminated string
- .byte b1, … , bn
- .double d1, …, dn
- .float f1, … , fn
- .word w1, …., wn
- .space n (in bytes)

System Calls

Service	System call code	Arguments	Result
print_int	1	a0 = integer	
print_float	2	fa0 = float	
print_double	3	fa0 = double	
print_string	4	a0 = address of string	
read_int	5		integer in a0
read_float	6		float in fa0
read_double	7		double in fa0
read_string	8	a0=address of buffer, a1=length of buffer	
exit	10		

See RARS Help for a full list of system calls

Using System Calls

- **Load the system call code into a7**
- **Load arguments into registers if needed**
- **Results will be in a0 if returning values**
- **e.g. print the integer in s0**

```
addi a7, zero, 1
add a0, s0, zero
ecall
```

Steps to Follow

- Open new file (star on paper icon)
- Enter your RISC-V code
- Save the file (use .s as file extension)
- Assemble (wrench)
- Run (green arrow)

Sample Program: Input/Output

```
.data

.text

main:
        addi a7, zero, 5          # get an integer from user
        ecall                     # integer in a0

        addi a7, zero, 1          # display integer in decimal
        ecall
```

Sample Program: Display Prompts

```
.data
        prompt: .asciz "Enter a number: “
        output: .asciz "The number you entered was: "
.text 0x00400000
.globl main
main:
        addi a7, zero, 4            # display prompt
        la a0, prompt
        ecall

        addi a7, zero, 5            # get number from user
        ecall
        add s0, a0, zero            # store 1st in s0

        addi a7, zero, 4            # display output
        la a0, output
        ecall

        add a0, s0, zero             # put number in a0
        addi a7, zero, 1             # and display it
        ecall

        addi a7, zero, 10           #code to exit cleanly
        ecall
```