Architecture COMP15111 Course Notes

Molshin Nurassyl

Contents

Week 1: Introduction to Computer Architecture

Introduction

- How components of a computer fit together to execute programs.
- The hardware programming model.
- Microarchitecture.
- System architecture.

Computer Architecture and Hardware Design

Bus: How the resources are organized

- How it is actually implemented.

RISC-V and ARM

- Can be implemented in different processors.
- Can run programmable hardware (FPGA).
- Can run in software (simulated).

Computer Architecture

Computer architecture cares about the high-level design and organization of the processor. It connects hardware and software together.

What is a Program?

- Apps, OS, Online Services, AI models.
- A sequence of unambiguously specified computer operations that collectively achieve a purpose.

How Does This Work?

$$\begin{aligned} \text{Users write} \rightarrow&\ \text{High Level Programs} \\ \text{Compiled into} \rightarrow&\ \text{Assembly} \\ \text{Assembled into} \rightarrow&\ \text{Machine Code} \\ \text{Which is executed} \rightarrow&\ \text{Hardware} \end{aligned}$$

Levels of Abstraction

1. **Higher Levels (Human)**
    - Easier to understand.
    - Can run on different systems.
    - Less control.
2. **Lower Levels (Machine Code)**
    - Precise control.
    - Specific for each computer family.
3. **Assembly Language (Compromise)**
    - Uses RISC-V.

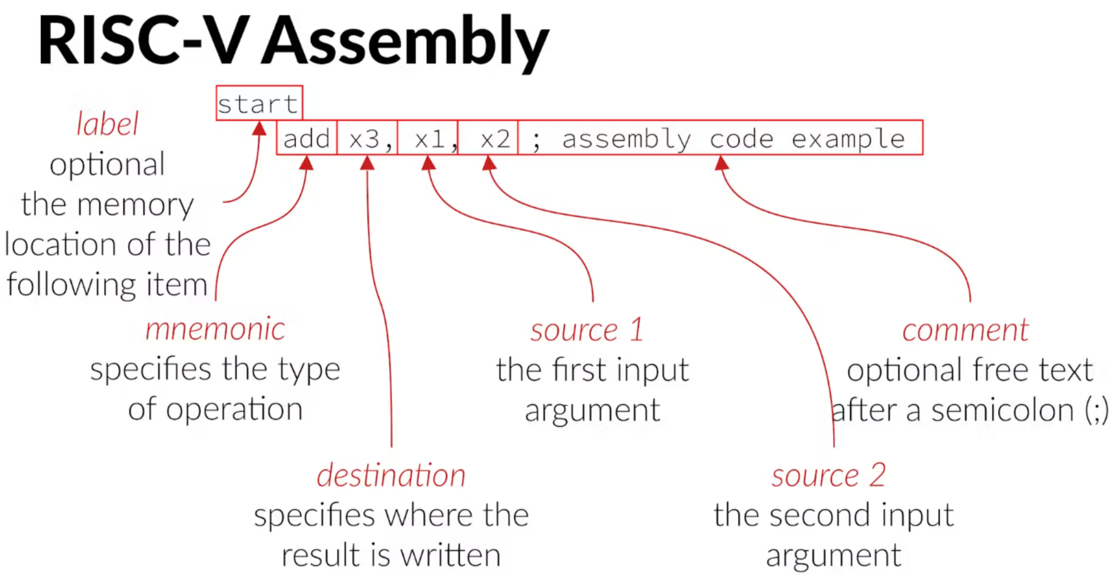

Figure 1: How instruction looks like

Registers

- Memory inside the CPU.
- Limited size but super fast.

Load-Store Architecture

- Operation operands are registers.
- Load instructions read data from memory into a register.
- Store instructions write data from a register into memory.
- More efficient use of memory.
- Simpler and faster instructions.

RISC-V

- 32 registers.
- 32-bit wide.
- $x0$ to $x31$.

The RISC-V Instruction Set

Data Processing Instructions

- Arithmetic, Logic, Shift, or Comparison.
- General format:

```
mnemonic rd, rs1, rs2 ; or
mnemonic rd, rs1, imm;
```

Data Processing Instructions

	Arithmetic	Logic	Shift	Comparison
Register only	add → Addition	and → Bitwise AND	sll → Shift left logical	
	sub → Subtraction	or → Bitwise OR	srl → Shift right logical	
	mul{..}→Multiply No option → lower half of 64-bit result h → upper half (signed) hu → upper half (unsigned) hsu → upper half (signed * unsigned)	xor → Bitwise XOR	sra → Shift right arithmetic	slt{u} → set if rs1 < rs2
	div{u} → Division (optional u indicates unsigned)			
	rem{u} → Remainder (optional u indicates unsigned)			
Immediate versions	addi	andi	slli	slti{u}
	~~subi~~ (pseudo)	ori	srli	
		xori	srai	

Figure 2: Data Processing instructions

Instruction Types

Comparison

$$\text{slt}\{x, y\} \rightarrow \text{Set if rs1 ¡ rs2}$$

Memory Operations

- Load: `lw rd, imm(rs)`
- Store: `sw rs2, imm(rs1)`

```
sum = a + b + c
lw x1, a
lw x2, b
add x3, x1, x2
```

Example Program

```
lw x3, c
add x4, x3, x2 ; (a + b) + c
sw x5, sum, x6
```

Week 2: Control Flow and Representation of Information

Control Flow

The Fetch Execute Cycle:

- **Initialisation** (PC contains the address of the first instruction, normally 0)
- **Fetch**
  - Read the word in the location pointed to by PC
  - Increment PC by 4
- **Execute**
  - Decode the instruction
  - Apply the operation on operands
  - Repeat forever
- **Execution is serial** (Going down the list of instructions)
- **Little flexibility**
  - Cannot reuse instructions without copy-pasting
  - Cannot change what is executed

Flow Control Instructions

Jump - Unconditional flow instructions
Branch - Conditional flow instructions

Unconditional Control Flow

- `jal rd, label`
  - Jump and Link
  - Next instruction stored in `rd`: `rd` $\leftarrow$ `PC+4`
  - Execution jumps to the address represented by `label`: `PC` $\leftarrow$ `label`
- `jalr rd, rs1, imm`
  - Jump and Link with register
  - Next instruction stored in `rd`: `rd` $\leftarrow$ `PC+4`
  - Execution jumps to the address represented by `rs1 + imm`: `PC` $\leftarrow$ `rs1 + imm`

Conditional Control Flow

General Syntax

- `beq cond, rs1, rs2, label`
- Branch if `cond`
- If `cond` is true for `rs1` and `rs2`: `PC` $\leftarrow$ `label`
- If `cond` is false: `PC` $\leftarrow$ `PC+4`

Conditions:

- `eq` = equal
- `ne` $\neq$ not equal

- `lt{u}` $<$ less than
- `le{u}` $\leq$ less or equal
- `gt{u}` $>$ greater
- `ge{u}` $\geq$ greater or equal

Branch Pseudo-Instructions:

- `blt` and `bgt` are symmetrical
- `blt x1, x2, target` $\Rightarrow$ `x1 < x2`   `bgt x2, x1, target` $\Rightarrow$ `x1 > x2`

Test for Zero

- `beqz rs=0`   `beq rs, x0, target`
- `blez rs` $\leq$ `0`
- `bgtz rs` $>$ `0`
- `bgez rs` $\geq$ `0`

Example:

- `beq x1, x2, label`   Jump to label if `x1 = x2`
- `bgez x1, label`   Jump to label if `x1` $\geq$ `0`

Representing Information

$2^n = 4$ (for $n = 2$ bits): 00, 01, 10, 11

Character Representation

Overall English $\Rightarrow$ total 81 values
(A-Z=26, a-z=26, 0-9=10, 17 symbols, +2 space and newline)
$\log_2(81) = 6.4 \approx 7$ bits
UTF-8 (Unicode/ISO 10646)

- A superset of ASCII
- Can use up to four bytes (32 bits)

Number Representation

ASCII is simple and compact but only for English
UTF-8 supports all possible languages
Strings are sequences of characters in memory (null-terminated strings)
Sets of bits are natively treated as base 2.
How many bits?

- More bits $\rightarrow$ better range
- Less bits $\rightarrow$ less storage

Python: doesn't care about storage
RISC-V: 8 bits (byte), 16 bits (half-word), 32 bits (word)
64-bit architecture also includes double-word (64 bits)

Base Conversion

Base-k: If we have N digits, they are numbered from 0 to N-1.
Can count k^N numbers.

Negative Numbers Two's Complement
Range: -2^{n-1} to $2^{n-1}-1$

- 8 bits: -128 to 127
- Algebraic approach: $-x$ is represented as $(2^N - x)$
- $-1 \Rightarrow 256 - 1 = 255 = 11111111$
- $-128 \Rightarrow 256 - 128 = 128 = 10000000$

Alternative Representation: Lower N-1 bits represent the positive number, bit N-1 represents its negative value $-(2^{N-1})$

Binary Logic: $-x$ is represented as the inverse of $x+1$

Unsigned Operations

- $x_1 = FFFFFFFF_{16}$ and $x_2 = 00000002_{16}$
- x_2 is always 2 (MSB is zero)
- x_1 is $2^{32}-1$ if unsigned, -1 if signed

Comparison:

- If unsigned: $x_1 = 2^{32}-1$ and $x_2 = 2 \Rightarrow x_1 > x_2$
- If signed: $x_1 = -1$ and $x_2 = 2 \Rightarrow x_1 < x_2$

Division:

- If unsigned: $x_1/x_2 \rightarrow 2^{31}-1 \rightarrow 0xFFFFFFFF$
- If signed: $x_1/x_2 \rightarrow 0 \rightarrow 0x00000000$

Week 3: RISC-V Memory and Instruction Encoding

RISC-V Memory

- Everything is 32-bits.
- Memory is byte addressable, while memory needs to use 32-bit numbers.

Data Types

- Other sizes supported for load/store:
  - Halfword → 16-bit, Byte → 8-bit
  - Load: `lh`, `lb` or `lhu`, `lbu`
  - Store: `sh`, `sb`
  - Sign doesn't matter

Little Endian

```
x1 = 0x12345678
sw x1, 12[x0] -> 0x12345678
lw x1, 12[x0] -> 0x00005678
lb x1, 12[x0] -> 0x00000078
```

Instruction Encoding

- Bits [6:0] is the opcode
- $2^7 = 128$ possible instructions
- All register arguments are 5-bits

R-Type

- Register-Register
- 2 inputs and 1 output register
- `funct7` and `funct3` → specific operation (e.g., sub, add)

command structure

- `rs1`, `rs2` → source register
- `rd` → destination register
- Opcode → instruction category

I-Type

- Register-Immediate
- 1 input and 1 output register operands
- 12 bits for immediate value

S-Type

- Store instructions
- 2 input register operands

B-Type (Branch)

- Immediate value is relative to PC: target = PC + imm

U-Type

- Long address instructions
- 1 destination register
- 20 bits for imm

J-Type (Jump and Link)

- PC + imm = target

Encoded Immediate Values

Two groups of encoded immediate values:

1. I/S/B → 12 bits (almost all instructions with immediate values)
2. U/J → 20 bits (e.g., `lui`, `auipc`, and `jal`)

I-Type (12-bit constant)

- Range: -2048 to 2047 or 0 to 4096
- Example: `addi x1, x1, 1000` (used for `addi`, `lw`, `jalr`)

B-Type (12-bit branch offset)

- Relative to the current PC
- Multiplied by 2, range: -4096 to 4095
- Examples: `beq`, `bne`, `blt`

Load/Store Immediate Values

- 12-bit offsets, range: -2048 to 2047
- Relative to the value in associated register
- Example: `lw x1, 1000[x2]` → address = `x2` + 1000

J-Type (20-bit offsets)

- Long unconditional jump
- PC-relative

U-Type

- Load 20-bit imm value in upper 20 bits of a register

Format	**Immediate Size**	**Immediates Positions**	**Usage**	**Range**
I-Type	12 bits	[31:20]	Constant value or offset for loads/arithmetic	-2048 to +2047
S-Type	12 bits	[31:25] (imm[11:5]) + [11:7] (imm[4:0])	Offset for store address	-2048 to +2047
B-Type	12 bits	[31] (imm[12]) + [30:25] (imm[10:5]) + [11:8] (imm[4:1]) + [7] (imm[11])	Offset for branch condition	-4096 to +4094
U-Type	20 bits	[31:12]	Upper 20 bits of a constant	0 to $2^{20} - 1$
J-Type	20 bits	[31] (imm[20]) + [30:21] (imm[10:1]) + [20] (imm[11]) + [19:12] (imm[19:12])	Offset for jump target	-2^{20} to $+2^{20} - 1$

Table 1: Immediate Value Types in RISC-V Instruction Formats

12-bit and 20-bit Immediates

- 12-bit immediates
  - Small values, local conditional branches
  - Accessing data from a specific memory region
- 20-bit immediates
  - Long unconditional jumps
  - Combined with 12-bit immediates to construct 32-bit addresses in two steps

Example Usage

```
WriteChar
    li x10, 'A'
    li x17, 0
    ecall
ReadChar
    li x17, 1
    ecall
WriteString
    la x10, msg
    li x17, 2
    ecall
Stop
    li x17, 5
    ecall
Decimal
    li x10, 123
    li x17, 3
    ecall
```

Week 4: [Title]

Integer Arithmetic

0.1 Memory Instructions

- lw : Load Word from memory
  Syntax lw rd, imm[rs]
- Pseudo lw: Load Word from label
- sw: Store Word from label
- Pseudo sw: Store Word to label

Week 5: [Title]

[Add Week 2 notes here.]

Week 7: [Title]

[Add Week 2 notes here.]

Week 8: [Title]

[Add Week 2 notes here.]