ECE 281

Lesson 14:

RV32I, R- and I-Type Instructions

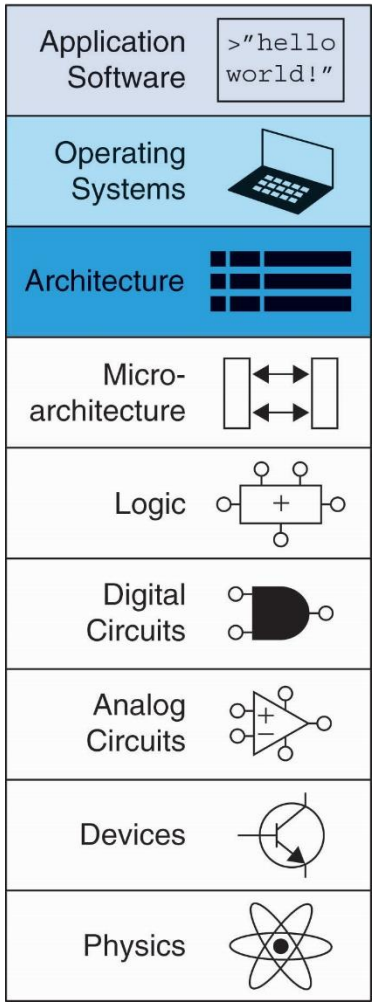
Digital Design and Computer Architecture Lecture Notes

Modified for use by USAF Academy, 2025

Chapter 6 :: Topics

- Introduction
- Assembly Language
- Programming
- Machine Language
- Addressing Modes
- Lights, Camera, Action: Compiling, Assembly, & Loading
- Odds & Ends

Videos
- Introduction
- Instructions
- Operands
- Immediates (Constants)
- Logical Instructions
- Machine Language: R-Type Instruction Formats



Appendix B: RV32I Instruction Set Summary.pdf (also front/back cover of your book!)

Assembly Language

- **Instructions:** commands in a computer's language
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)
- **RISC-V architecture:**
  - Pronounced "Risk Five"
  - Developed by Krste Asanovic, David Patterson and their colleagues at UC Berkeley in 2010.
  - First widely accepted open-source computer architecture

Once you've learned one architecture, it's easier to learn others (ECE 382 → ARM)

Instructions

Instructions: Addition

C Code

```
a = b + c;
```

RISC-V assembly code

```
add a, b, c
```

- **`add:`** mnemonic indicates operation to perform
- **`b, c:`** source operands (on which the operation is performed)
- **`a:`** destination operand (to which the result is written)

Instructions: Subtraction

Similar to addition - only **mnemonic** changes

C Code

```
a = b - c;
```

RISC-V assembly code

```
sub a, b, c
```

- `sub:` mnemonic
- `b, c:` source operands
- `a:` destination operand

Chapter 6: Architecture

Operands

Operands

- **Operand location:** physical location in computer
  - Registers
  - Memory
  - Constants (also called *immediates*)

Operands: Registers

We will go in-depth into how to make a register in later lessons ☺

- RISC-V has 32 32-bit registers
- Registers are faster than memory
- RISC-V called “32-bit architecture” because it operates on 32-bit data

Smaller is Faster

- RISC-V includes only a small number of registers

RISC-V Register Set

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

Homework

Operands: Registers

- **Registers:**
  - Can use either name (i.e., `ra`, `zero`) or `x0`, `x1`, etc.
  - Using name is preferred
- Registers used for **specific purposes**:
  - `zero` always holds the **constant value 0**.
  - the ***saved registers***, `s0–s11`, used to hold variables
  - the ***temporary registers***, `t0–t6`, used to hold intermediate values during a larger computation
  - ***arguments*** are passed in `a0–a7` & **returned** in `a0–a1`
  - Discuss others later; until we get to function calls just use `t0–t6`

Instructions with Registers

- Revisit `add` instruction

C Code

```
a = b + c;
```

RISC-V assembly code

```
# s0 = a, s1 = b, s2 = c
add s0, s1, s2
```

# indicates a single-line comment

Instructions with Constants

- `addi` instruction

C Code

```
a = b + 6;
```

RISC-V assembly code

```
# s0 = a, s1 = b
addi s0, s1, 6
```

Generating Constants

Generating 12-Bit Constants

- 12-bit signed constants (immediates) using `addi`:

C Code

```
// int is a 32-bit signed word
int a = -372;
int b = a + 6;
```

RISC-V assembly code

```
# s0 = a, s1 = b
addi s0, zero, -372
addi s1, s0, 6
```

Any immediate that needs **more than 12 bits** cannot use this method.

Generating 32-bit Constants

- Use load upper immediate (`lui`) and `addi`
- `lui`: puts an immediate in the upper 20 bits of destination register and 0's in lower 12 bits

C Code

```
int a = 0xFEDC8765;
```

RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC8
addi s0, s0, 0x765
```

Remember that `addi` **sign-extends** its 12-bit immediate

Load Immediate Pseudoinstrcution

Table B.7 RISC-V pseudoinstructions

Pseudoinstruction	RISC-V Instructions	Description	Operation
`nop`	`addi x0, x0, 0`	no operation	
`li rd,` $imm_{11:0}$	`addi rd, x0,` $imm_{11:0}$	load 12-bit immediate	`rd =` $SignExtend(imm_{11:0})$
`li rd,` $imm_{31:0}$	`lui rd,` $imm_{31:12}$ * `addi rd, rd,` $imm_{11:0}$	load 32-bit immediate	`rd =` $imm_{31:0}$

Chapter 6: Architecture

Logical / Shift Instructions

Logical Instructions: Example 1

Source Registers

s1	0100 0110	1010 0001	1111 0001	1011 0111
s2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly Code

```
and s3, s1, s2
or  s4, s1, s2
xor s5, s1, s2
```

Result

s3	0100 0110	1010 0001	0000 0000	0000 0000
s4	1111 1111	1111 1111	1111 0001	1011 0111
s5	1011 1001	0101 1110	1111 0001	1011 0111

Logical Instructions: Example 2

Source Values

t3	0011	1010	0111	0101	0000	1101	0110	1111
imm	1111	1111	1111	1111	1111	1010	0011	0100

sign-extended

Assembly Code

```
andi s5, t3, -1484
ori  s6, t3, -1484
xori s7, t3, -1484
```

Result

s5								
s6								
s7								

-1484 = **0xA34** in 12-bit 2's complement representation.

Shift Instructions

Shift amount is in (lower 5 bits of) a register

- `sll`: shift left logical
  - **Example:** `sll t0, t1, t2 # t0 = t1 << t2`
- `srl`: shift right logical
  - **Example:** `srl t0, t1, t2 # t0 = t1 >> t2`
- `sra`: shift right arithmetic
  - **Example:** `sra t0, t1, t2 # t0 = t1 >>> t2`

Immediate Shift Instructions

Shift amount is an immediate between 0 to 31

- `slli`: shift left logical immediate
  - **Example:** `slli t0, t1, 23 # t0 = t1 << 23`
- `srli`: shift right logical immediate
  - **Example:** `srli t0, t1, 18 # t0 = t1 >> 18`
- `srai`: shift right arithmetic immediate
  - **Example:** `srai t0, t1, 5 # t0 = t1 >>> 5`

Chapter 6: Architecture

Machine Language

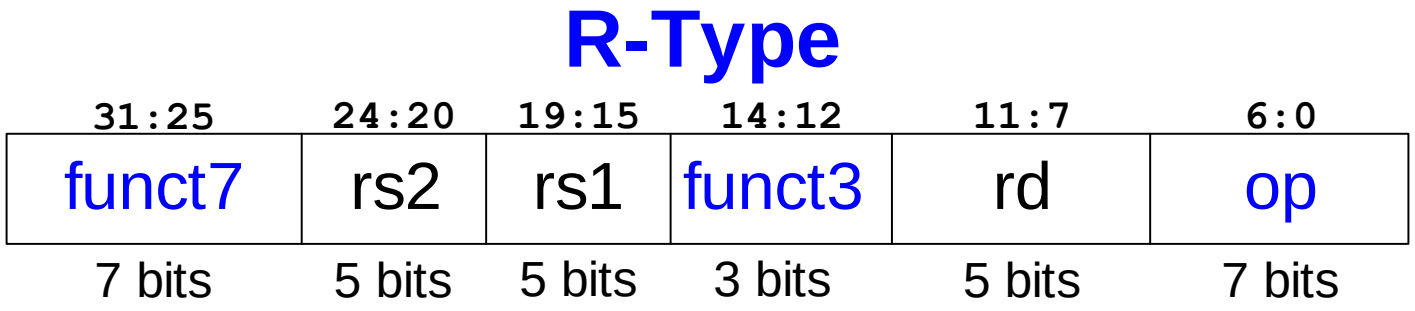
Machine Language

You will need to be able to write machine code, but not yet

- Binary representation of instructions
- Computers only understand 1's and 0's
- 32-bit instructions
  - Simplicity favors regularity: 32-bit data & instructions
- **4 Types of Instruction Formats:**
  - R-Type
  - I-Type
  - S/B-Type
  - U/J-Type

R-Type

- ***Register-type***
- 3 register operands:
  - `rs1`, `rs2`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the *operation code* or *opcode*
  - `funct7`, `funct3`: the *function* (7 bits and 3-bits, respectively) with opcode, tells computer what operation to perform



Instruction Fields & Formats

Instruction	op	funct3	Funct7	Type
`add`	0110011 (51)	000 (0)	0000000 (0)	R-Type
`sub`	0110011 (51)	000 (0)	0100000 (32)	R-Type
`and`	0110011 (51)	111 (7)	0000000 (0)	R-Type
`or`	0110011 (51)	110 (6)	0000000 (0)	R-Type
`addi`	0010011 (19)	000 (0)	-	I-Type
`beq`	1100011 (99)	000 (0)	-	B-Type
`bne`	1100011 (99)	001 (1)	-	B-Type
`lw`	0000011 (3)	010 (2)	-	I-Type
`sw`	0100011 (35)	010 (2)	-	S-Type
`jal`	1101111 (111)	-	-	J-Type
`jalr`	1100111 (103)	000 (0)	-	I-Type
`lui`	0110111 (55)	-	-	U-Type

See Appendix B for other instruction encodings

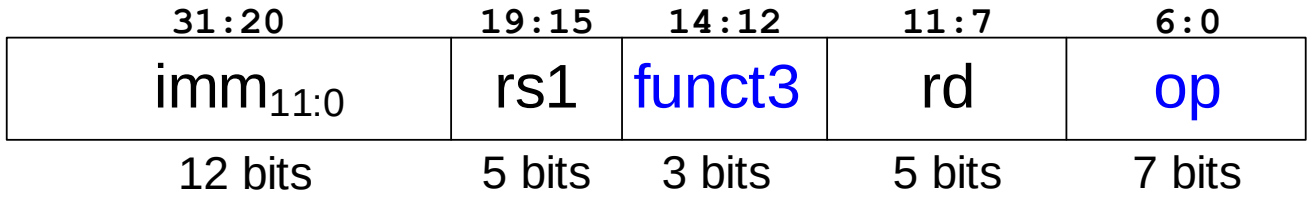
R-Type Examples

Assembly	Field Values						Machine Code						
	funct7	rs2	rs1	funct3	rd	op	funct7	rs2	rs1	funct3	rd	op	
`add s2, s3, s4` `add x18,x19,x20`	0	20	19	0	18	51	0000 000	10100	10011	000	10010	011 0011	(0x01498933)
`sub t0, t1, t2` `sub x5, x6, x7`	32	7	6	0	5	51	0100 000	00111	00110	000	00101	011 0011	(0x407302B3)
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

I-Type

- ***Immediate-type***
- 3 operands:
  - `rs1`: register source operand
  - `rd`: register destination operand
  - `imm`: 12-bit two's complement immediate
- Other fields:
  - `op`: the opcode
    - Simplicity favors regularity: all instructions have opcode
  - `funct3`: the function (3-bit function code)
    - with opcode, tells computer what operation to perform

I-Type



I-Type Examples

Assembly	$imm_{11:0}$	rs1	funct3	rd	op	Machine Code					
`addi s0, s1, 12` `addi x8, x9, 12`	12	9	0	8	19	0000 0000 1100	01001	000	01000	001 0011	**(0x00C48413)**
`addi s2, t1, -14` `addi x18,x6, -14`	-14	6	0	18	19	1111 1111 0010	00110	000	10010	001 0011	**(0xFF230913)**
`lw t2, -6(s3)` `lw x7, -6(x19)`	-6	19	2	7	3	1111 1111 1010	10011	010	00111	000 0011	**(0xFFA9A383)**
`lh s1, 27(zero)` `lh x9, 27(x0)`	27	0	1	9	3	0000 0001 1011	00000	001	01001	000 0011	**(0x01B01483)**
`lb s4, 0x1F(s4)` `lb x20,0x1F(x20)`	0x1F	20	0	20	3	0000 0001 1111	10100	000	10100	000 0011	**(0x01FA0A03)**
	12 bits	5 bits	3 bits	5 bits	7 bits	12 bits	5 bits	3 bits	5 bits	7 bits	

Design Principle 4

Good design demands good compromises

- Multiple instruction formats allow flexibility
  - `add`, `sub`: use 3 register operands
  - `lw`, `sw`, `addi`: use 2 register operands and a constant
- Number of instruction formats kept small
  - to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster).

Practice

Venus RV32I Simulator

```
.text
# Example for register inspection
addi t0, t0, 2016 # init t0 to 2016

ecall # Exit
```

Exercise 6.7 The `nor` instruction is not part of the RISC-V instruction set because the same functionality can be implemented using existing instructions. Write a short assembly code snippet that has the following functionality: `s3 = s4` NOR `s5`. Use as few instructions as possible.

```
.text

li s4, s4, 0b1010 # init s4
li s5, s5, 0b1100 # init s5
# your code here

ecall # Exit
```