Topics for Chapter 2

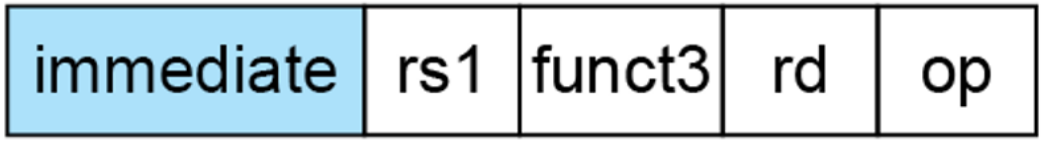
- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS
  - Memory access instructions (Chapters 2.3)
  - Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part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

Addressing Modes

- **How values are found ("addressed") to be used within operations**
  - Immediate addressing
  - Register addressing
  - Base addressing
  - PC-relative addressing

Immediate Addressing

1. Immediate addressing

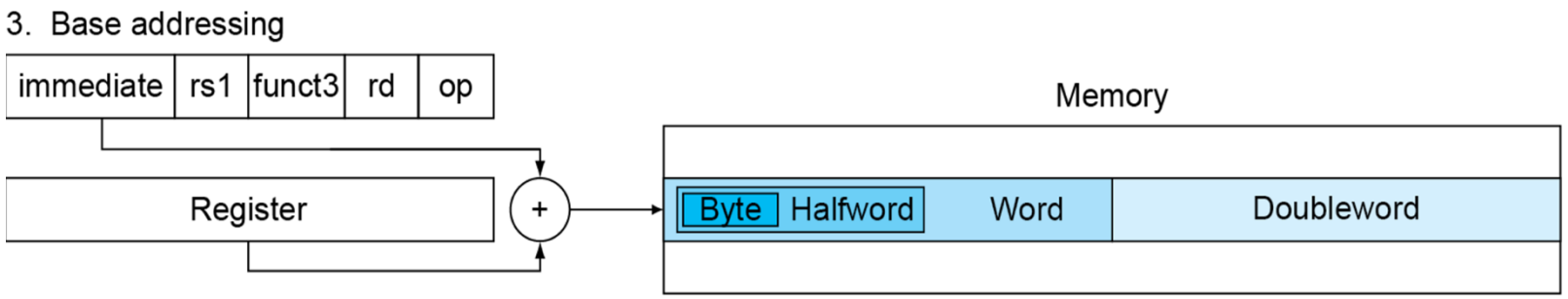


- **Value is directly in the instruction**
  - Always signed
  - Sign extend
- **What instruction format is this?**
- **What instructions use this?**
- **Which other instruction formats have immediates in them?**

Register Addressing

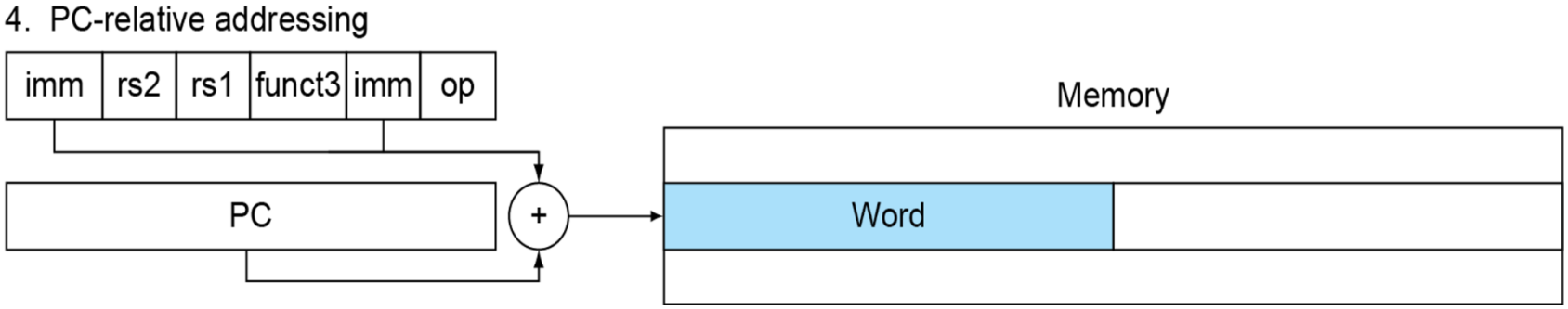
- Value is in a register, the register number is in the instruction
- What instruction format is this?
- Is the value in the register signed or unsigned?
- What other formats have register addressing?

Base Addressing



- **Register contents and immediate value are added to find address in memory**
  - Register has base address, immediate is offset from base
- **Are reg/imm values signed or unsigned?**
- **What instruction format is this?**
- **What instructions use this?**

PC-relative Addressing



- Immediate value is added to PC to produce address in memory
- Is Immediate value signed or unsigned?
- What is expected at that address?
- What instruction format is this?
- What other formats use this?

Decoding Machine Language

Decoding Machine Language

- **Machine language ⇒ Assembly language ⇒ C**
- **Decoding steps:**
  - Look at `opcode`: 51 means R-Format
  - Use instruction type to determine which fields exist.
  - Write out RISC-V assembly code, converting each field to name, register number/name, or decimal/hex number.
  - Logically convert this RISC-V code into valid C code (decompile)
    - Compress expressions to 1 expression (with parentheses)
    - Compress if/goto to loops, jal/jalr to functions

Decoding Example

- 6 machine language instructions in hexadecimal:

```
0x00006533
          0x00D05863
     0x00C50533
          0xFFF68693
      0xFF5FF06F
```

- first instruction at address 0x00400000

Decoding Step 1: Convert Hex to Binary

```
0000 0000 0000 0000 0110 0101 0011 0011
0000 0000 1101 0000 0101 1000 0110 0011
0000 0000 1100 0101 0000 0101 0011 0011
1111 1111 1111 0110 1000 0110 1001 0011
1111 1111 0101 1111 1111 0000 0110 1111
```

Decoding Step 2: Identify Format

```
0000 0000 0000 0000  0110  0101   0011 0011
0000 0000 1101  0000  0101 1000   0110 0011
0000 0000 1100  0101  0000  0101  0011 0011
1111 1111   1111   0110  1000  0110  1001 0011
1111 1111   0101   1111  1111   0000  0110 1111
```

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	opcode
I	imm[11:0]			rs1	funct3	rd	opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

Decoding Step 3: Separate Fields

R	0	0	0	6	10	0x3F
SB	8 (After putting bits in order)	13	0	5	Look at other immediate field	0x63
R	0	12	10	0	10	0x3F
I	0xFFFFFFFF		13	0	13	0x13
UJ	0xFFFFA (After putting bits in right order)				0	0x6F

have not added missing bit to SB or UJ, just untangled them to their stored values.

Decoding Step 4: Disassemble to RISC-V Instructions

Address	Assembly instruction
0x00400000	or x10, x0, x0
0x00400004	bge x0, x13, 0x10 #4 instructions forward
0x00400008	add x10, x10, x12
0x0040000C	addi x13, x13, 0xFFFFFFFF
0x00400010	jal x0, 0xFFFFF4 #add to PC for destination add
0x00400014	

values for jal and beq now include hidden bit (but are PC-relative)

Decoding Step 4b. Make RISC-V Code Better

- **Fix the branch/jump and add labels, registers**

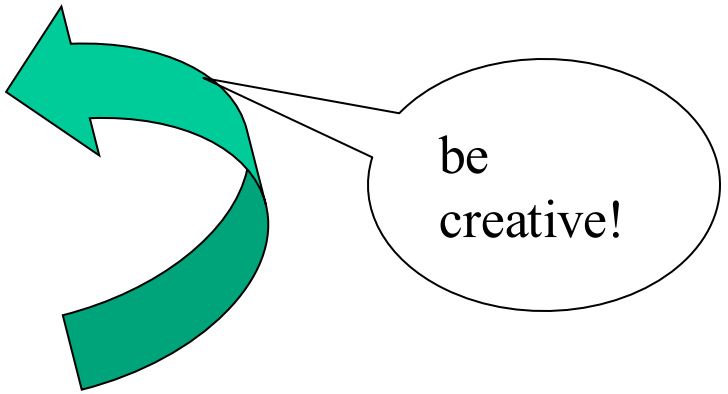
	`or a0, zero, zero`
`Loop:`	`bge zero, a3, Exit`
	`add a0, a0, a2`
	`addi a3, a3, -1`
	`jal zero, Loop`
`Exit:`	

Decoding Step 5: Decompile to C Code

a0: product; a2: multiplicand; a3: multiplier

C Code:

```
product = 0;
while (multiplier > 0) {
        product += multiplicand;
        multiplier -= 1;
}
```



Machine Code:

```
0x00006533
0x00D05863
0x00C50533
0xFFF68693
0xFF5FF06F
```

Assembly Code:

```
 or a0, zero, zero
Loop: bge zero, a3, Exi
     add a0, a0, a2
           addi a3, a3, -
1
           jal zero, Loo
Exit:
```

Conclusion

- Simplifying RISC-V: Define instructions to be same size as data word (one word) so that they can use the same memory (compiler can use `lw` and `sw`).
- Computer actually stores programs as a series of these 32-bit numbers.
- RISC-V machine language instruction: 32 bits