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)

Stored-Program Concept

- **Computers are built on 2 key principles:**
  1) Instructions are represented as numbers
  2) Entire programs can be stored in memory to be read or written just like numbers (data).
- **Simplifies SW/HW of computer systems:**
  - Memory technology for data is also used for programs

Consequence #1: Everything Addressed

- **Since all instructions and data are stored in memory as numbers, everything (instructions, data words) has a memory address:**
  - both branches and jumps use these
- **C pointers are just memory addresses: point to anything in memory**
  - Unconstrained use of addresses can lead to nasty bugs
- **One register keeps address of instruction being executed: "Program Counter" (PC)**
  - a pointer to memory
  - Intel: Instruction Address Pointer

Consequence #2: Binary Compatibility

- **Programs are distributed in binary form**
  - Programs bound to specific instruction set
  - Different version for Macintoshes and PCs
- **New machines want to run old programs ("binaries") as well as programs compiled to new instructions**
- **Leads to instruction set evolving over time**
- **that latest PCs still use 80x86 instruction set (Pentium 4);**
  - Could still run program from 1981 PC today
  - Selection of Intel 8086 in 1981 for 1st IBM PC
- **By treating the instructions in the same way as the data, a stored-program machine can easily change the instructions**
  - the machine is reprogrammable.
  - a program can easily increment or modify the address portion of instructions

Instructions as 32-bit Numbers

- **Currently all data we work with is in words (32-bit blocks):**
  - Each register is a word.
  - `lw` and `sw` both access memory one word at a time.
- **RISC-V wants simplicity**
  - since data is in words, make instructions be words too

RISC-V Instructions Formats

- **Divide 32-bit instruction word into "fields".**
  - Each field tells something about instruction.
- **6 basic types of instruction formats in RISC-V**
  - R-format: ("register")
    - math and logic: add, sub, and, or, sll, sra
  - I-format: ("immediate")
    - lw, lb, slli, srli, addi, andi, ori, xori, jalr
  - S-format ("store")
    - sw, sb
  - SB-format ("branch"), i.e., B format
    - beq, bne, bge, blt, bgeu, bltu
  - UJ-format ("jump"), i.e., J format
    - jal
  - U-format (instructions with "upper immediates")
    - lui, auipc

RISC-V Instruction Formats

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

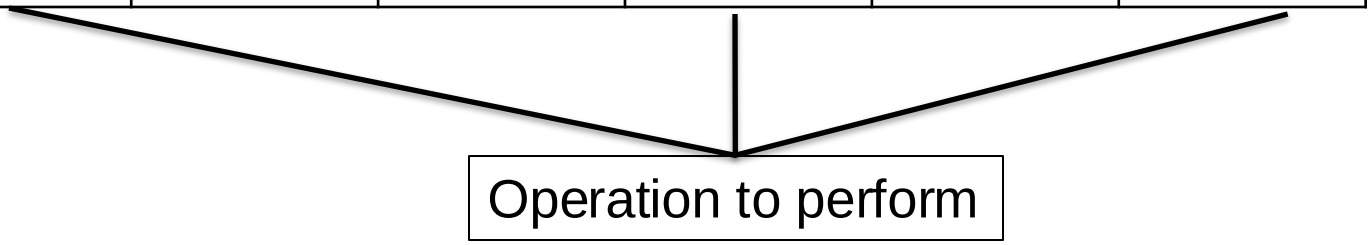
- **Small number of formats**
- **Similar layouts**
- **Why?**

You are the Compiler and Assembler

- **For this course you have been the compiler**
  - You turn what into what?
- **Now you're going to be the assembler**
  - The assembler turns assembly into what?

R-format

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
	2nd source register	1st source register		Destination register	



- **R-format instructions**
  - add, sub, and, or, xor, sll, srl, sra...
  - e.g., add rd, rs1, rs2

R-format Example

```
add t0, s1, s2
```

- **For simplicity, let's use decimal to fill the fields first**

funct7	rs2	rs1	funct3	rd	opcode
0	18	9	0	5	51

- **Convert the decimal fields to binary**

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
funct7	rs2	rs1	funct3	rd	opcode
0000000	10010	01001	000	00101	0110011

- **Convert that to Hexadecimal: 0x__ __ __ __ __ __ __ __**

- **Why are all the register fields 5 bits?**

Fields of R-Format Instructions

funct7	rs2	rs1	funct3	rd	opcode

- **Specify instruction:**
  - `opcode`: 51 (0110011) for all R-Format instruction
  - `Funct7` combined with funct3: exactly specifies the instruction
  - Why aren't opcode and funct a single 14-bit field?
- **Operands: 5 bits fits unsigned numbers 0 .. 31**
  - `rs1` (source register): first operand
  - `rs2` (source register): second operand
  - `rd` (destination register): register receiving result of computation

Exercise

Consider `add t5, s0, s1`. This is an R-type instruction, so it will have 6 fields:

funct7	rs2	rs1	funct3	rd	opcode

- **What decimal value is stored in field rs2?**

- **Of the 32 bits in the instruction, how many bits indicate that the instruction must perform an add operation?**

Exercise

- **Which RARS instruction does the following combination of fields represent? All values are in decimal.**

0x005302B3=0000 0000 0101 0011 0000 0010 1011 0011
Opcode=0110011
Rs1=00110, Rs2=00101, RD=00101
Funct3=000 Funct7=0000000

0	5	6	0	5	51
a) `sub t0, t1, t1`					
b) `add t0, t0, t1`					
c) `sub t0, t1, t0`					
d) `add t0, t1, t0`					

I-Format (Immediate)

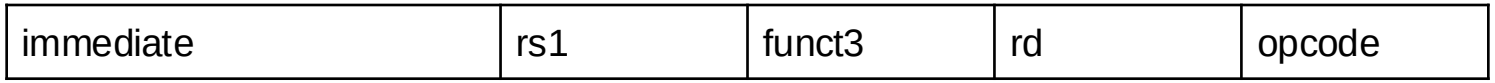
- **I stands for “Immediate”**
- **I-Format instructions**
  - lw, lb, slli, srli, addi, andi, ori, xori, jalr

I-Format

immediate [11:0]	rs1 [4:0]	funct3 [2:0]	rd [4:0]	opcode [6:0]

- **Instructions with immediate values**
  - instruction has opcode, no function
  - instruction has only two registers
  - immediate value limited to 12 bits
- **What fields are the same as R-Format?**

I-Format fields



- **`opcode/funct3`: specifies instruction**
- **`rs1`: the *only* register operand (if there is one)**
  - when isn't there a register operand?
- **`rd`: register will "receive" result of computation**
  - target fits, for I-Format
- **`immediate`: 12 bit value**
  - range of values is?
  - *always sign-extended!*

I-Format instructions

- **addi: R[rd] = R[rs1]+ SignExt (immed)**
- **Convert `addi s2, s1, 5` to binary**

12 bits	5 bits	3 bits	5 bits	7 bits
immediate	rs1	funct3	rd	opcode

	immediate	rs1	funct3	rd	opcode
decimal	5	9	3	18	19
binary	0000 0000 0101	0 1001	000	1 0010	001 0011
hex					

- **Which register corresponds to bits 19:15?**

Other I-format Instructions

- **What are other I-format instructions that we commonly use?**
  - `andi, ori, xori, srli, slli`
- **What's the unusual I-format instruction?**
  - `lw!`
  - `R[rd]={MEM[R[rs1]+signExt (immed)](31:0)}`
- **Convert `lw t0, 32(s3)` to a machine instruction**
- **12-bit immediate has a range of _____**

Shift Instructions

Word of caution: slli, srli, and srai are a little different

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

One of the higher order immediate bits is used to differentiate srli and srai

The shamt only uses 5 bits because you can only shift by a maximum of 31 positions (Adds bounds checking)

While considered “I-type”, the format is similar to R-type

R Format vs I Format

- How does the hardware tell the difference between `add` and `addi`?

S-Format Instructions

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- **Used for stores (sw, sb, sh)**
- **e.g., sw rs2, immed (rs1)**
- **Want to keep rs2, rs1 in same place as other instructions**
- **opcode/funct3: specifies instruction**
- **rs1: the first register operand**
- **rs2: the second register operand**
- **immediate: 12-bit value when combined**
  - Why did they split up the immediate value?

sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]

What If Immediate Won't Fit in 12 bits?

- **Solution:**
  - Handle it in software - make the compiler do it!
  - Introduce two new instructions: lui, auipc
  - Both instructions can take 20-bit immediate

U-Format Instructions: lui

- **lui: load upper immediate**
  - lui rd, immediate
  - R[rd] = {20-bit immediate, 12b'0}
- **lui writes the upper 20 bits of rd with the immediate value, and clears the lower 12 bits**
- **Together with an addi to set low 12 bits, can create any 32-bit value in a register using two instructions (lui/addi)**

```
lui x10, 0x87654          # x10 = 0x87654000
addi x10, x10, 0x321      # x10 = 0x87654321
```

Corner Case

- How to set oxDEADBEEF?

lui x10, 0xDEADB #x10 = 0xDEADB000

addi x10, x10, 0xEEF #x10 = 0xDEADAEEF

Incorrect!

Why?

1. 0xEEF is out of range

2. How about change it to 0xFFFFFEEF?

In addi, 12-bit immediate is always sign-extended

If top bit of the 12-bit immediate is a 1, it will subtract 1 from upper 20 bits

Corner Case Solution

- How to set oxDEADBEEF?

lui x10, 0xDEADC #x10 = 0xDEADC000

addi x10, x10, 0xFFFFFEEF #x10 = 0xDEADBEEF

Pre-increment value placed in upper 20 bits, if sign bit will be set on immediate in lower 12 bits.

```
li t1, 0xDEADBEEF  # pseudo instruction
```

Expands to:

```
lui t1, 0xFFFDEADC
addi t1, t1, 0xFFFFFEEF
```

Compiler takes care of calculating values as shown above

U-Format Instructions: auipc

- **auipc: add upper immediate to PC and place in rd (don't change PC)**
  - auipc rd, immediate
  - rd = PC + {immediate, 12b'0}

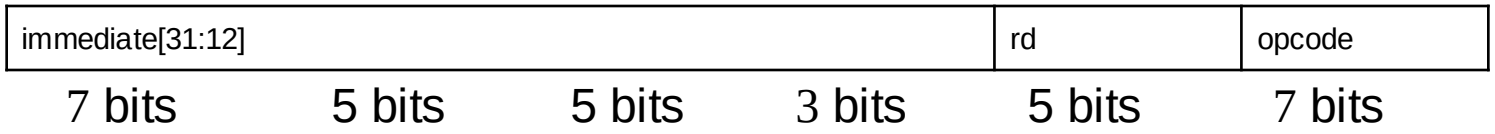
- **Pseudo instruction "la a0, LABEL" becomes**

  auipc a0, IMMEDIATE

  addi a0, a0, ADJUST

  IMMEDIATE and ADJUST calculated by assembler based on PC and LABEL

U-Format fields



- **`opcode`: specifies instruction**
- **`rd`: destination register**
- **`immediate`: 20-bit value**

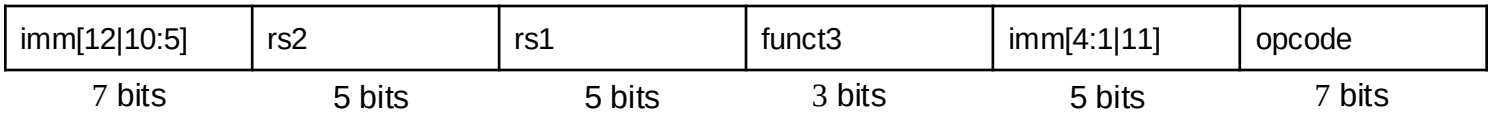
`lui`	Load Upper Imm	U	`0110111`			`rd = imm << 12`
`auipc`	Add Upper Imm to PC	U	`0010111`			`rd = PC + (imm << 12)`

SB/B Format Instructions

imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

beq	SB	Branch EQual	if(R[rs1]==R[rs2) PC=PC+{imm,1b'0}
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}
blt	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}

SB/B Format



- **e.g., beq rs1, rs2, Label**
- **funct3: operation of the instruction**
  - beq= ____, bne=____
- **rs1 and rs2: registers to compare**
- **immediate: see Reference Card**
  - RISC-V supports compressed half-word (16 bits) instructions
  - To accommodate this, all branch offsets (the immediate) are half-word aligned
  - Range is $\pm 2^{11}$ half words, or $\pm 2^{10}$ words
  - What happened to bit-0 of the immediate?

Branch Immediates

- **12 bits, shifted left one bit (multiply by ____)**
  - why?
- **sign extended**
- **added to PC (the branch instruction's address)**
- **Can branch _____ words from PC (_____ bytes)**
- **Why?**
  - branches are used by: `if-else`, `while`, `for`
  - branch often changes PC by a small amount
  - loops are generally small (typically ~50 instructions)

Branch Calculation

- **If we *don't* take the branch:**
  - `PC = PC + 4` (byte address of next instruction)
- **If we *do* take the branch:**
  - `PC = PC + (immediate * 2)`
- **Note:**
  - `immediate` field specifies the number of halfwords to jump
  - `immediate` field is a two's complement value (signed)
  - add `immediate` shifted left 1 bit to PC
  - range:
    - PC + 2*(-2^_____) … PC + 2*(2^_____-1)

Branch Example

opcode=_____

rs1=_____

rs2=_____

immediate=_____

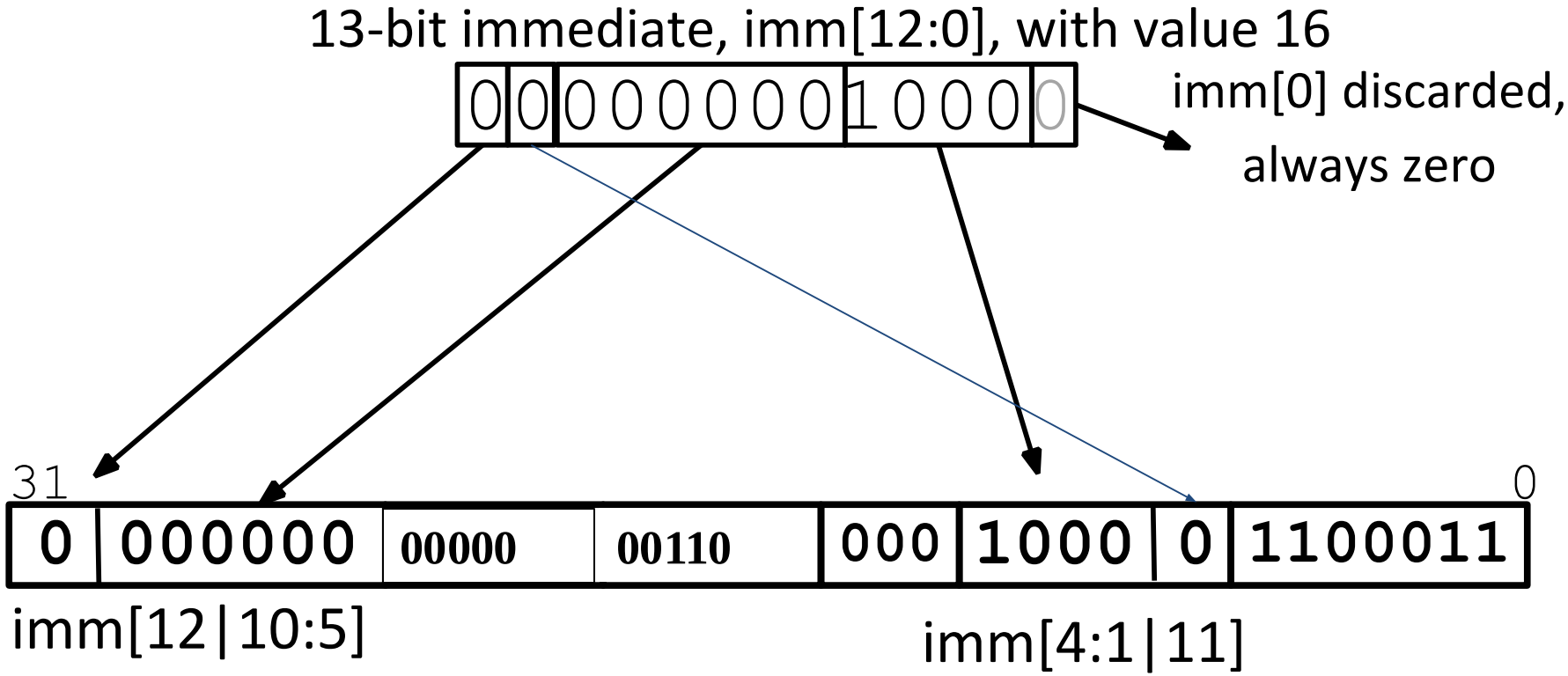
```
Loop:
    beq  t1, zero, End
    add  t0, t0, t2
    addi t1, t1, -1
    jal  zero, Loop
End:
```

- **Branch offset = 4 32-bit instructions = 16 bytes**
- **immediate is number of instructions*2 to add to (<0 to subtract) address of branch instruction.**

imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
???????	00000	00110	000	?????	1100011

Branch Example

beq t1, zero, offset = 16 bytes



hex 0x __ __ __ __ __ __ __ __

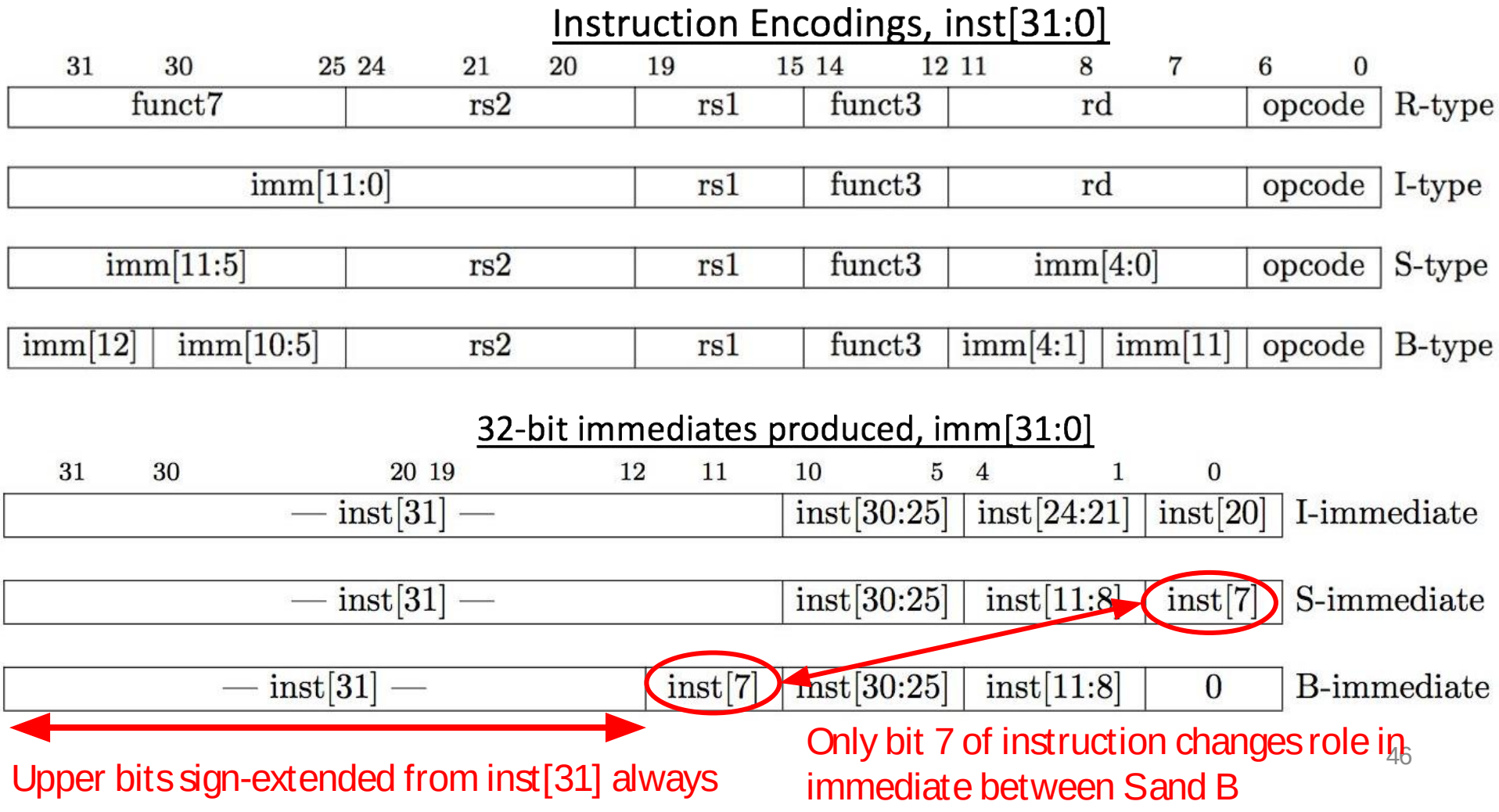
Why are the Addresses so Tangled up for SB?

imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode

- **Because shifting costs space on silicon.**
- **With this layout, only bit 11 has to be shifted, and a 0 put in its place, relative to the S-format immediate value handling**

- **Why not put bit 12 down in the 0 position?**
  - Make it easy for sign extension

Why is it so Confusing?!



Unconditional Branches/Jumps

- **Conditional branches**
  - assume we won't branch very far
  - can specify changes in PC
- **Unconditional branch (jal)**
  - may jump *anywhere* in memory (a 32-bit address)
  - can't fit both a 7-bit opcode and a 32-bit address in the 32-bit instruction
  - compromise (good design)

UJ-Format (i.e., J Format)

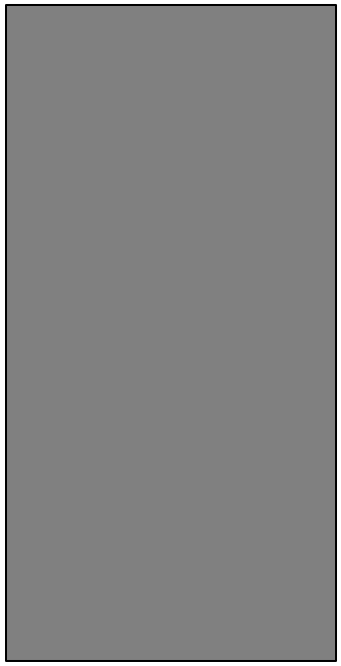
immediate[20|10:1|11|19:12]	rd	opcode

jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}

- **opcode format identical to R-format and I-format**
  - consistency simplifies hardware
- **Rest of instruction used for target address**
  - RISC-V supports compressed half-word (16 bits) instructions
  - To accommodate this, all branch offsets (the immediate) are half-word aligned (all addresses are even … bit 0 is always 0)
  - What happened to bit-0 of the immediate?
- **How is address calculated? Untangled and shift left 1 bit**

How Far Can jal Jump?

- **Range is _____ instructions**
  - 2^20 values since 20 bits
  - *2 since lowest bit is an assumed 0 to get # bytes
  - /4 since 4 bytes per instruction
- **Where in that range do we start? (trick question)**



- **Jump only inadequate if needs to straddle a 2^{18} boundary. It works 99.999…% of the time, since programs usually aren't that long.**
- **Technically cannot jump *anywhere* in memory.**

UJ-Format Example

```
0x00400000 Loop: slli t1, s3, 2
0x00400004       …
0x00400008       …
0x0040000C       …
0x00400010       …
0x00400014 jal zero, Loop
0x00400018       …
```

opcode = _____

PC = _____

PC@Loop = _____

PC=PC+{imm,1b'0}=0x00400014+0xFFFEC (2's complement)

0xFFFEC=0x1 1111 1111 1111 1110 1100 (21 bits)

immediate[20|10:1|11|19:12]	rd	opcode
11111110110111111111	00000	1101111

Don't stereotype!

- **Why isn't jalr a UJ-type instruction?**

- **What is the difference between `jal` and `jalr`?**

Branching Far Away

- **What if conditional branch must jump further?**

```
beq t0, t1, L1   # L1 >> 2^10-1
```

can become:

```
beq t0, t1, L2
    ...
  jal zero, L3 # jump around the re-jump
L2: jal zero, L1 # within 2^18 from here
L3: ...          # continue with code
```

All Instruction Formats

- **6 basic types of instruction formats in RISC-V**
  - R-format: (“register”)
    - math and logic: add, sub, and, or, sll, sra
  - I-format: (“immediate”)
    - lw, lb, slli, srli, addi, andi, ori, xori, jalr
  - S-format (“store”)
    - sw, sb
  - SB-format (“branch”) or B-format
    - beq, bne, bge, blt, bgeu, bltu
  - UJ-format (“jump”) or J-format
    - jal
  - U-format (instructions with “upper immediates”)
    - lui, auipc

RISC-V Instruction Format Summary

- **Minimize number of instructions**
  - 6 types
  - Instructions have 1, 2, or 3 operands

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

After-Class Activities

- **Take a few instructions from your last program and generate the machine language for them by hand.**
- **Compare your results to the assembled results in RARS.**

Topics for Chapter 2

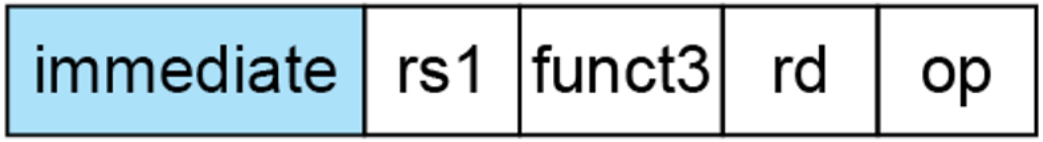
- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS
  - Memory access instructions (Chapters 2.3)
  - Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 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

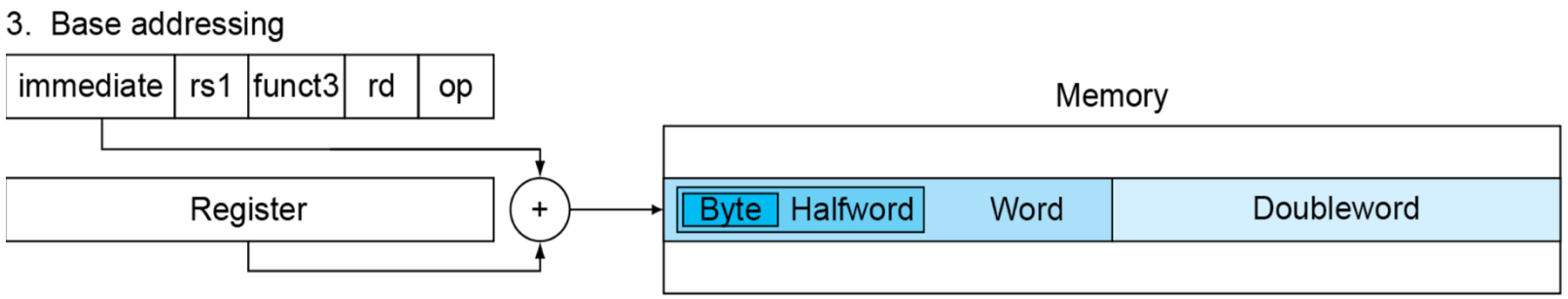


- **Value is directly in the instruction**
  - Always signed
  - Sign extend
- **What instruction format is this?**
- **What instructions use this?**
- **Which other instruction formats have immediates in them?**

Register Addressing

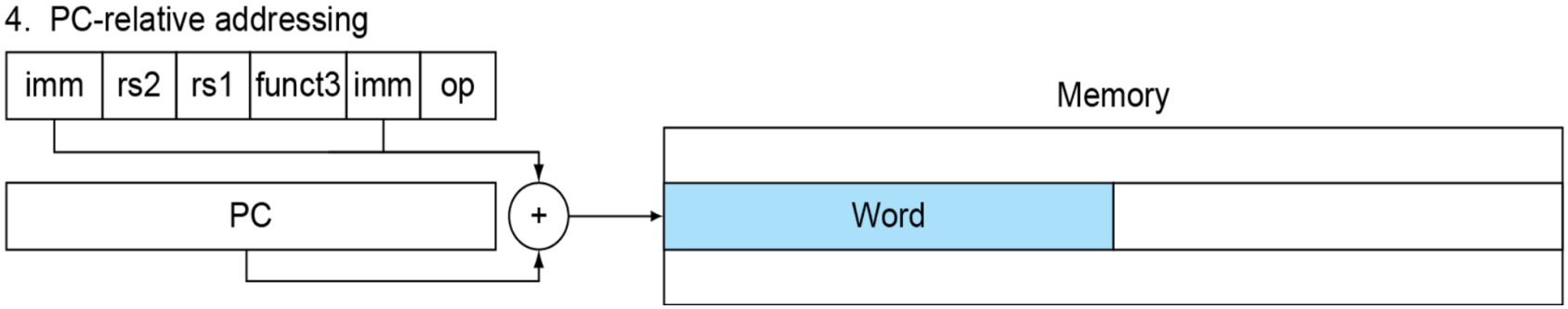
- Value is in a register, the register number is in the instruction
- What instruction format is this?
- Is the value in the register signed or unsigned?
- What other formats have register addressing?

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**

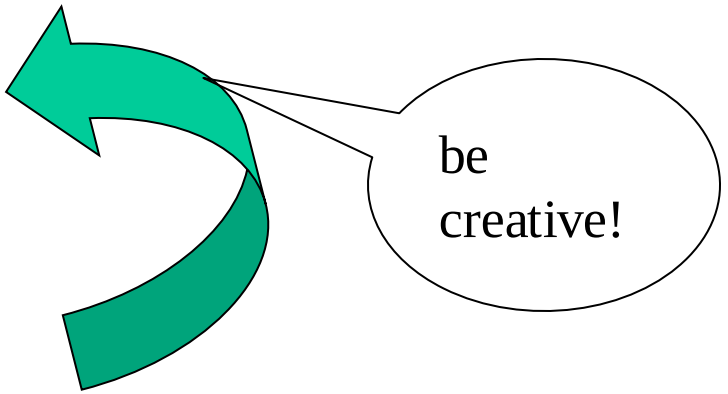
	`or a0, zero, zero`
`Loop:`	`bge zero, a3, Exit`
	`add a0, a0, a2`
	`addi a3, a3, -1`
	`jal zero, Loop`
`Exit:`	

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**