



18-447 Lecture 3:

RISC-V Instruction Set Architecture

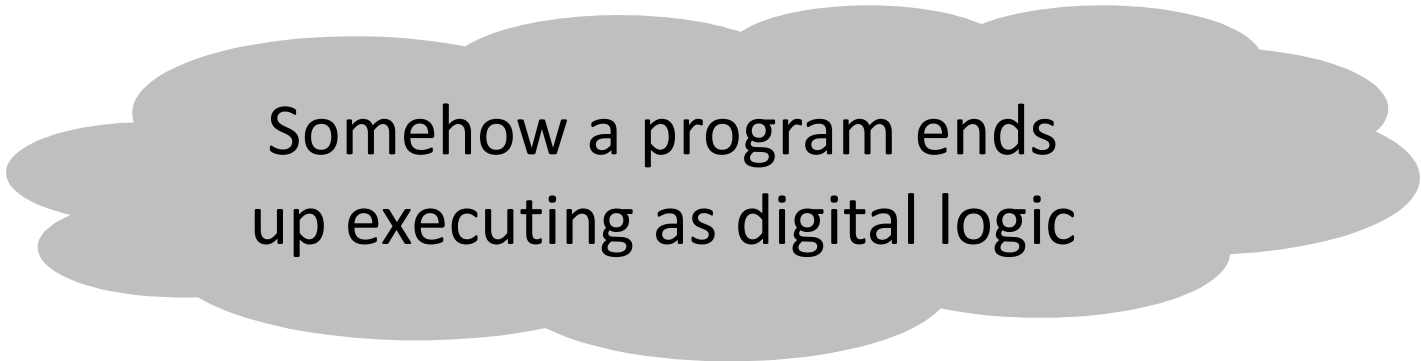
James C. Hoe

Department of ECE

Carnegie Mellon University

What is 18-447?

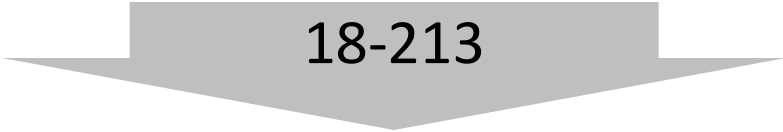
- 18-213: Introduction to Computer Systems
 - “C” as the model of computation
 - interact with the computer hardware through OS
 - what about the details below the **abstraction**?



Somehow a program ends
up executing as digital logic

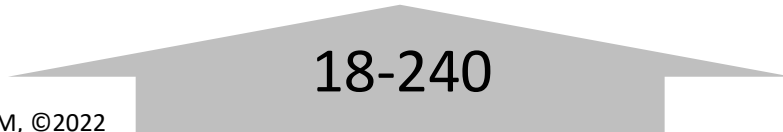
- 18-240: Fundamentals of Computer Engineering
 - digital logic as the model of computation
 - gates and wires as building blocks
 - what about the details below this abstraction?

18-447: Fuzzy to Concrete



18-213

- “Computer Architecture”
 - functional spec for software and programmers
 - design spec for the hardware people
- Computer Organization
 - take architecture to “micro”architecture
 - how to assemble/evaluate/tune
- Computation Structures
 - digital representations
 - processing, storage and I/O elements



18-240

Housekeeping

- Your goal today
 - get bootstrapped on RISC-V RV32I to start Lab 1
(will return to visit general ISA issues on Jan 31)
- Notices
 - Check Canvas and Piazza regularly
 - Student survey (on Canvas), **due next Wed**
 - H02: Lab 1, Part A, **due Week 3**
 - H03: Lab 1, Part B, **due Week 4**
- Readings
 - P&H Ch2
 - P&H Ch4.1~4.4 (**Lecture 4 next time**)

What we mean by “architecture”?

How to specify what a computer does?

- Architectural Level

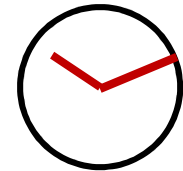


a clock has an hour hand and a minute hand,



a computer does?????....

You can read a clock without knowing how it works



conceptual

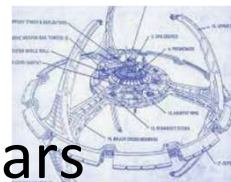
- Microarchitecture Level (think blueprint)



a particular clockwork has a certain set of gears arranged in a certain configuration



a particular computer design has a certain datapath and a certain control logic



physical

- Realization Level



machined alloy gears vs stamped sheet metal



CMOS vs ECL vs vacuum tubes



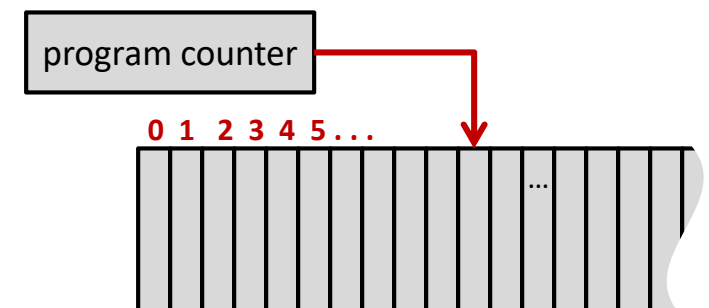
Stored Program Architecture

a.k.a. von Neumann

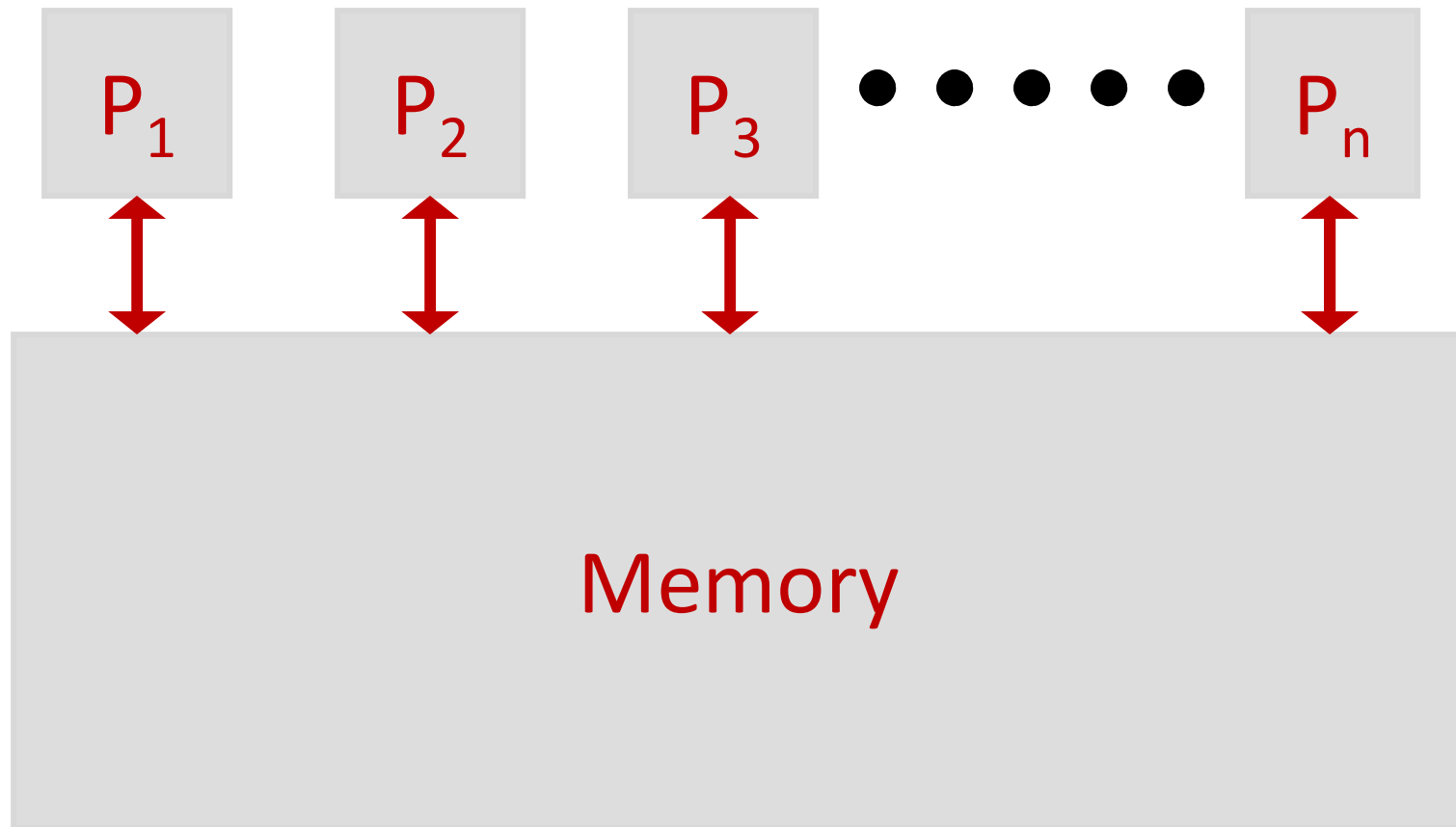


- Memory holds both program and data
 - instructions and data in a linear memory array
 - instructions can be modified as data
- Sequential instruction processing
 1. **program counter (PC)** identifies current instruction
 2. fetch instruction from memory
 3. update some state (e.g. **PC** and memory) as a function of current state according to instruction
 4. repeat

Dominant paradigm since its conception

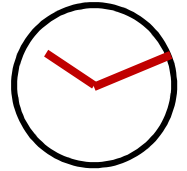


Non-von Neumann Architecture Example



Parallel Random-Access Machine

Very Different Architectures Exist



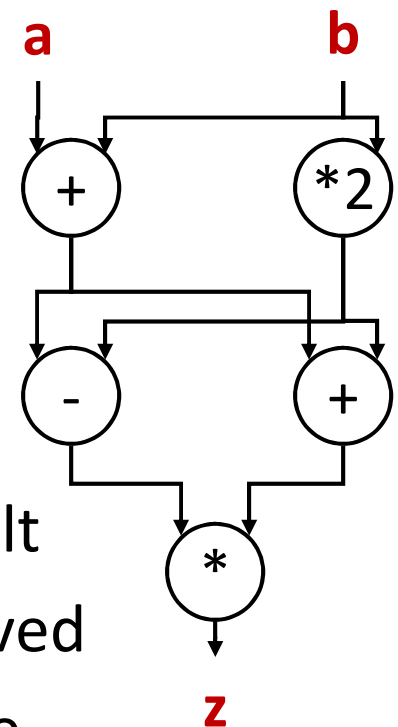
- Consider a von Neumann program
 - what is the significance of the instruction order?
 - what is the significance of the storage locations?

00:00

```

v := a + b ;
w := b * 2 ;
x := v - w ;
y := v + w ;
z := x * y ;

```



- Dataflow program instruction ordering implied by data dependence
 - instruction specifies who receives the result
 - instruction executes when operands received
 - no program counter, no intermediate state

[dataflow figure and example from Arvind]

Instruction Set Architecture (ISA): A Concrete Specification

“ISA” in a nut shell

- A stable programming target (to last for decades)
 - binary compatibility for SW investments
 - permits adoption of foreseeable technology

Better to compromise immediate optimality for future scalability and compatibility

- Dominant paradigm has been “von Neumann”
 - program visible state: memory, registers, PC, etc.
 - instructions to modified state; each prescribes
 - which state elements are read
 - which state elements—including PC—updated
 - how to compute new values of update state

Atomic, sequential, in-order

3 Instruction Classes (as convention)

- Arithmetic and logical operations
 - fetch operands from specified locations
 - compute a result as a function of the operands
 - store result to a specified location
 - update PC to the next sequential instruction
- Data “movement” operations (no compute)
 - fetch operands from specified locations
 - store operand values to specified locations
 - update PC to the next sequential instruction
- Control flow operations (affects only PC)
 - fetch operands from specified locations
 - compute a branch condition and a target address
 - if “branch condition is true” then $PC \leftarrow \text{target address}$
else $PC \leftarrow \text{next seq. instruction}$

Complete “ISA” Picture

- User-level ISA
 - state and instructions available to user programs
 - single-user abstraction on top a “virtualization”

For this course and for now, RV32I of RISC-V
- “Virtual Environment” Architecture
 - state and instructions to control virtualization (e.g., caches, sharing)
 - user-level, but for need-to-know uses
- “Operating Environment” Architecture
 - state and instructions to implement virtualization
 - privileged/protected access reserved for OS



system
arch

RV32I Program Visible State

program counter

32-bit “byte” address of current instruction

M[0]
M[1]
M[2]
M[3]
M[4]
M[N-1]

****note**** $x_0=0$

x1

x2

general purpose
register file

32x 32-bit words
named $x_0 \dots x_{31}$

32-bit memory “byte” address:
 2^{32} by 8-bit locations (4 GBytes)
 (there is some magic going on)

Register-Register ALU Instructions

- Assembly (e.g., register-register addition)

ADD rd, rs1, rs2

- Machine encoding

0000000	rs2	rs1	000	rd	0110011
7-bit	5-bit	5-bit	3-bit	5-bit	7-bit

- Semantics
 - $\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs1}] + \text{GPR}[\text{rs2}]$
 - $\text{PC} \leftarrow \text{PC} + 4$
- Exceptions: none (ignore carry and overflow)
- Variations
 - Arithmetic: {ADD, SUB}
 - Compare: {signed, unsigned} x {Set if Less Than}
 - Logical: {AND, OR, XOR}
 - Shift: {Left, Right-Logical, Right-Arithmetic}

Reg-Reg Instruction Encodings

31	25	24	20	19	15	14	12	11	7	6	0		
funct7			rs2		rs1		funct3		rd		opcode		R-type
0000000			rs2		rs1		000		rd		0110011		ADD
0100000			rs2		rs1		000		rd		0110011		SUB
0000000			rs2		rs1		001		rd		0110011		SLL
0000000			rs2		rs1		010		rd		0110011		SLT
0000000			rs2		rs1		011		rd		0110011		SLTU
0000000			rs2		rs1		100		rd		0110011		XOR
0000000			rs2		rs1		101		rd		0110011		SRL
0100000			rs2		rs1		101		rd		0110011		SRA
0000000			rs2		rs1		110		rd		0110011		OR
0000000			rs2		rs1		111		rd		0110011		AND

32-bit R-type ALU

[The RISC-V Instruction Set Manual]

Assembly Programming 101

- Break down high-level program expressions into a sequence of elemental operations
- E.g. High-level Code

```
f = ( g + h ) - ( i + j )
```

- Assembly Code
 - suppose f, g, h, i, j are in r_f, r_g, r_h, r_i, r_j
 - suppose r_{temp} is a free register

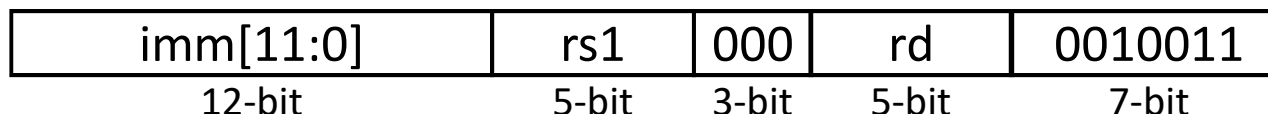
```
add r_temp r_g r_h    # r_temp = g+h
add r_f r_i r_j        # r_f = i+j
sub r_f r_temp r_f     # f = r_temp - r_f
```

Reg-Immediate ALU Instructions

- Assembly (e.g., reg-immediate additions)

ADDI rd, rs1, imm₁₂

- Machine encoding



- Semantics
 - $\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs1}] + \text{sign-extend}(\text{imm})$
 - $\text{PC} \leftarrow \text{PC} + 4$
- Exceptions: none (ignore carry and overflow)
- Variations
 - Arithmetic: {ADDI, ~~SUBI~~}
 - Compare: {signed, unsigned} x {Set if Less Than Imm}
 - Logical: {ANDI, ORI, XORI}
 - **Shifts by unsigned imm[4:0]: {SLLI, SRLI, SRAI}

Reg-Immediate ALU Inst. Encodings

31	20	19	15	14	12	11	7	6	0	
imm[11:0]					rs1	funct3	rd	opcode		I-type
imm[11:0]					rs1	000	rd	0010011		ADDI
imm[11:0]					rs1	010	rd	0010011		SLTI
imm[11:0]					rs1	011	rd	0010011		SLTIU
imm[11:0]					rs1	100	rd	0010011		XORI
imm[11:0]					rs1	110	rd	0010011		ORI
imm[11:0]					rs1	111	rd	0010011		ANDI

sign-extended immediate

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

unsigned

matches

32-bit I-type ALU

R-type encoding

Note: SLTIU does unsigned compare with sign-extended immediate


[The RISC-V Instruction Set Manual]

Load-Store Architecture

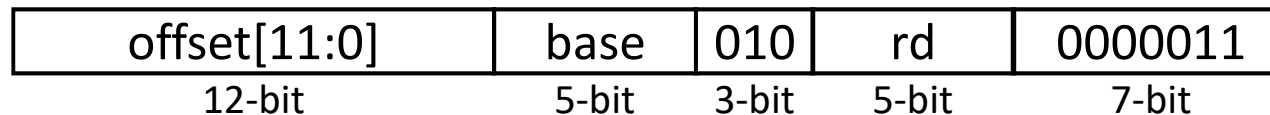
- RV32I ALU instructions
 - operates only on register operands
 - next PC always PC+4
- A distinct set of load and store instructions
 - dedicated to copying data between register and memory
 - next PC always PC+4
- Another set of “control flow” instructions
 - dedicated to manipulating PC (branch, jump, etc.)
 - does not effect memory or other registers

Load Instructions

- Assembly (e.g., load 4-byte word)

LW rd, offset₁₂(base) ←  rs1

- Machine encoding



- Semantics

- $\text{byte_address}_{32} = \text{sign-extend}(\text{offset}_{12}) + \text{GPR}[\text{base}]$
- $\text{GPR}[\text{rd}] \leftarrow \text{MEM}_{32}[\text{byte_address}]$
- $\text{PC} \leftarrow \text{PC} + 4$

- Exceptions: none for now

- Variations: LW, LH, LHU, LB, LBU

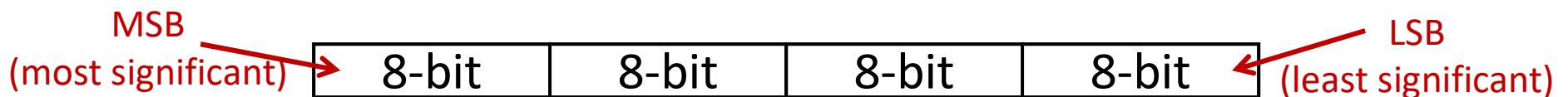
e.g., LB :: $\text{GPR}[\text{rd}] \leftarrow \text{sign-extend}(\text{MEM}_8[\text{byte_address}])$

LBU :: $\text{GPR}[\text{rd}] \leftarrow \text{zero-extend}(\text{MEM}_8[\text{byte_address}])$

RV32I is byte-addressable, little-endian (*until v20191213*)

When data size > address granularity

- 32-bit signed or unsigned integer word is 4 bytes
- By convention we “write” MSB on left



- On a byte-addressable machine

Big Endian

MSB			LSB
byte 0	byte 1	byte 2	byte 3
byte 4	byte 5	byte 6	byte 7
byte 8	byte 9	byte 10	byte 11
byte 12	byte 13	byte 14	byte 15
byte 16	byte 17	byte 18	byte 19

pointer points to the **big end**

Little Endian

MSB			LSB
byte 3	byte 2	byte 1	byte 0
byte 7	byte 6	byte 5	byte 4
byte 11	byte 10	byte 9	byte 8
byte 15	byte 14	byte 13	byte 12
byte 19	byte 18	byte 17	byte 16

pointer points to the **little end**

- What difference does it make?

check out **htonl()**, **ntohl()** in in.h

Load/Store Data Alignment

MSB	byte-3	byte-2	byte-1	byte-0	LSB
	byte-7	byte-6	byte-5	byte-4	

- Access granularity not same as addressing granularity
 - physical implementations of memory and memory interface optimize for natural alignment boundaries (i.e., return an aligned 4-byte word per access)
 - unaligned loads or stores would require 2 separate accesses to memory
- Common for RISC ISAs to disallow misaligned loads/stores; if necessary, use a code sequence of aligned loads/stores and shifts
- RV32I (until v20191213) allowed misaligned loads/stores but warns it could be very slow; if necessary, . . .

Store Instructions

- Assembly (e.g., store 4-byte word)

SW **rs2**, **offset**₁₂(**base**)

- Machine encoding

offset[11:5]	rs2	base	010	ofst[4:0]	0100011
7-bit	5-bit	5-bit	3-bit	5-bit	7-bit

- Semantics
 - $\text{byte_address}_{32} = \text{sign-extend}(\text{offset}_{12}) + \text{GPR}[\text{base}]$
 - $\text{MEM}_{32}[\text{byte_address}] \leftarrow \text{GPR}[\text{rs2}]$
 - $\text{PC} \leftarrow \text{PC} + 4$
- Exceptions: none for now
- Variations: SW, SH, SB

e.g., SB:: $\text{MEM}_8[\text{byte_address}] \leftarrow (\text{GPR}[\text{rs2}])[7:0]$

Assembly Programming 201

- E.g. High-level Code

```
A[ 8 ] = h + A[ 0 ]
```

where **A** is an array of integers (4 bytes each)

- Assembly Code

- suppose $\&A$, h are in r_A , r_h
- suppose r_{temp} is a free register

```
LW  rtemp 0(rA)      # rtemp = A[0]
add rtemp rh rtemp    # rtemp = h + A[0]
SW  rtemp 32(rA)      # A[8] = rtemp
                                # note A[8] is 32 bytes
                                #      from A[0]
```

Load/Store Encodings

- Both needs 2 register operands and 1 12-bit immediate

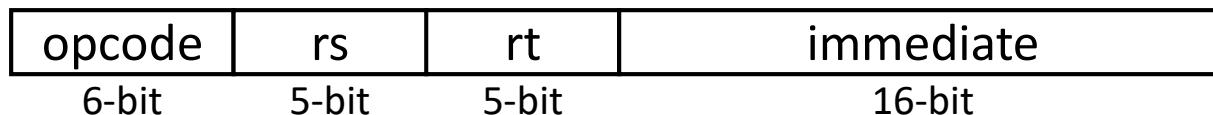
31	20	19	15	14	12	11	7	6	0	
imm[11:0]					rs1	funct3	rd	opcode		I-type
imm[11:0]					rs1	000	rd	0000011		LB
imm[11:0]					rs1	001	rd	0000011		LH
imm[11:0]					rs1	010	rd	0000011		LW
imm[11:0]					rs1	100	rd	0000011		LBU
imm[11:0]					rs1	101	rd	0000011		LHU

31	25	24	20	19	15	14	12	11	7	6	0		
imm[11:5]			rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[11:5]			rs2		rs1		000		imm[4:0]		0100011		SB
imm[11:5]			rs2		rs1		001		imm[4:0]		0100011		SH
imm[11:5]			rs2		rs1		010		imm[4:0]		0100011		SW

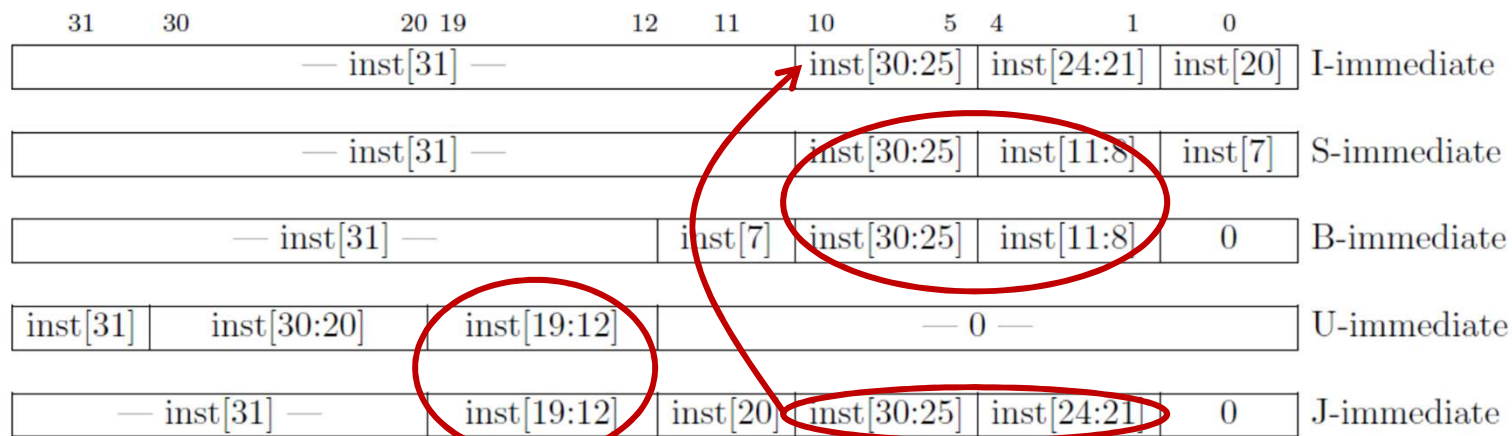
[The RISC-V Instruction Set Manual]

RV32I Immediate Encoding

- Most RISC ISAs use 1 register-immediate format

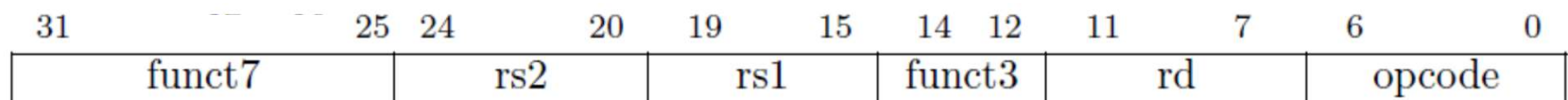


- rt field used as a source (e.g., store) or dest (e.g., load)
 - also common to opt for bigger 16-bit immediate
- RV32I adopts 2 different register-immediate formats (I vs S) to keep rs2 operand at inst[24:20] always
- RV32I encodes immediate in non-consecutive bits

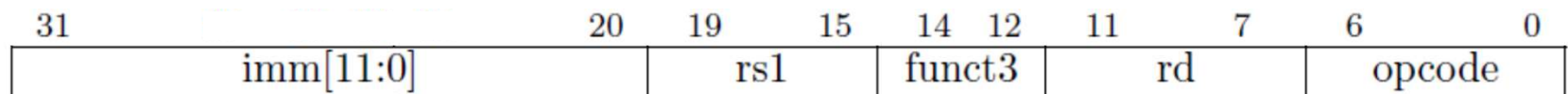


RV32I Instruction Formats

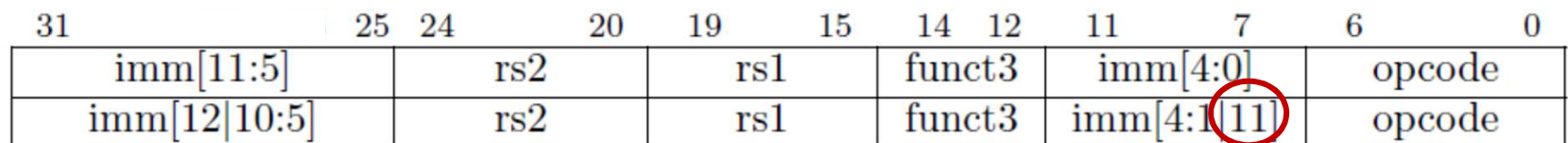
- All instructions 4-byte long and 4-byte aligned in mem
- R-type: 3 register operands



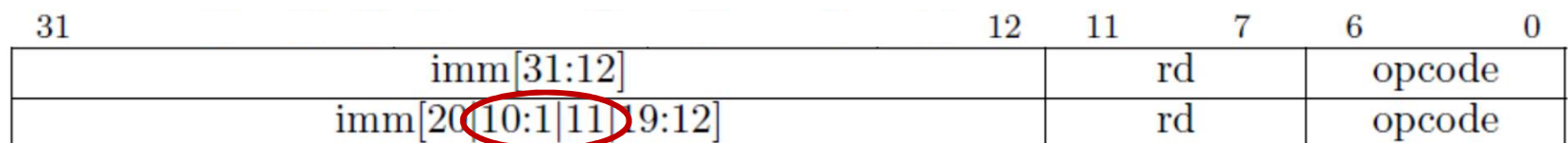
- I-type: 2 register operands (with dest) and 12-bit imm



- S(B)-type: 2 register operands (no dest) and 12-bit imm



- U(J)-type, 1 register operand (dest) and 20-bit imm



Aimed to simplify decoding and field extraction

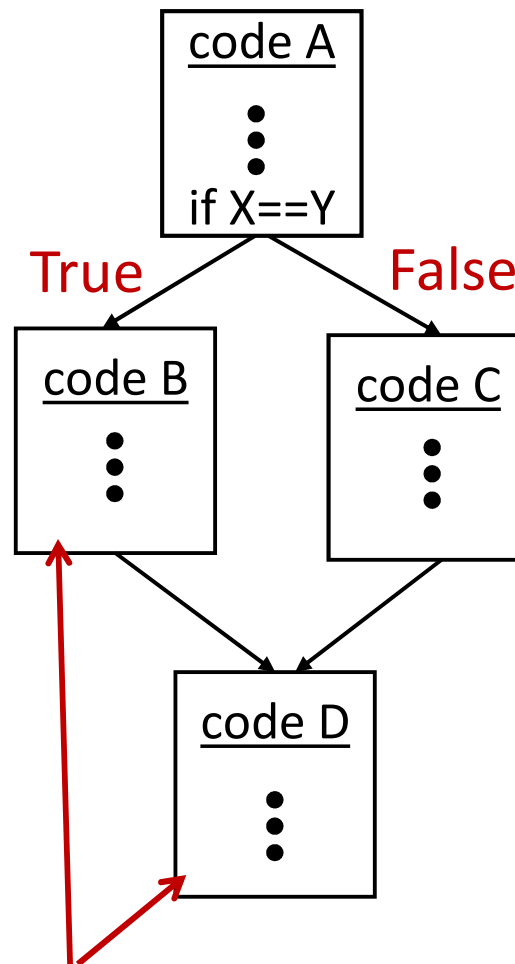
Control Flow Instructions

- C-Code

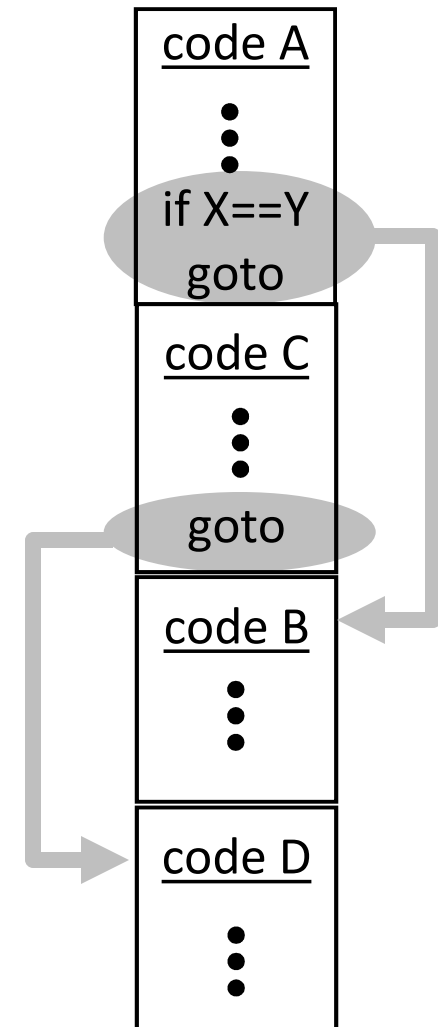
```

{ code A }
if X==Y then
    { code B }
else
    { code C }
{ code D }
  
```

Control Flow Graph



Assembly Code
(linearized)



basic blocks (1-way in, 1-way out, all or nothing)

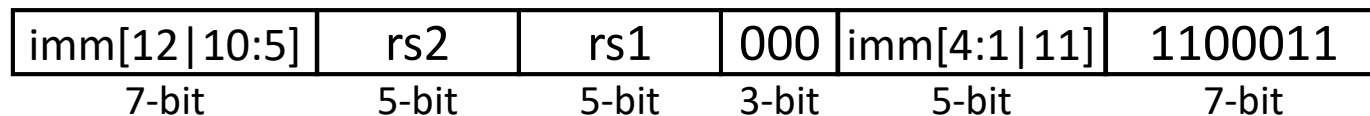
(Conditional) Branch Instructions

- Assembly (e.g., branch if equal)

BEQ *rs1*, *rs2*, *imm*₁₃

Note: implicit *imm*[0]=0

- Machine encoding



- Semantics

– target = PC + sign-extend(*imm*₁₃)

– if GPR[*rs1*]==GPR[*rs2*] then PC ← target
 else PC ← PC + 4

How far can you jump?

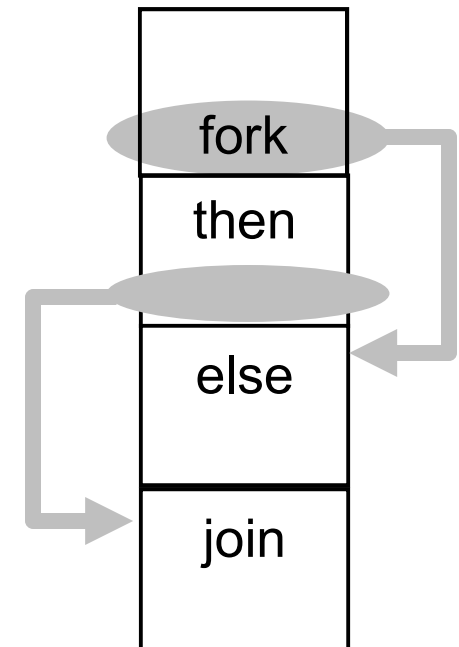
- Exceptions: misaligned target (4-byte) if taken
- Variations
 - BEQ, BNE, BLT, BGE, BLTU, BGEU

Assembly Programming 301

- E.g. High-level Code

```

if (i == j) then
    e = g
else
    e = h
f = e
  
```



- Assembly Code

– suppose e, f, g, h, i, j are in $r_e, r_f, r_g, r_h, r_i, r_j$

<code>bne r_i r_j L1</code>	<code># L1 and L2 are addr labels</code>
	<code># assembler computes offset</code>
<code>add r_e r_g x0</code>	<code># e = g</code>
<code>beq x0 x0 L2</code>	<code># goto L2 unconditionally</code>
<code>L1: add r_e r_h x0</code>	<code># e = h</code>
<code>L2: add r_f r_e x0</code>	<code># f = e</code>

Assembly Programming 302

- If you write C code:

```
for (int i=0; i<16; i++) {
    sum+=A[i];
}
```

- GCC -O generates code for:

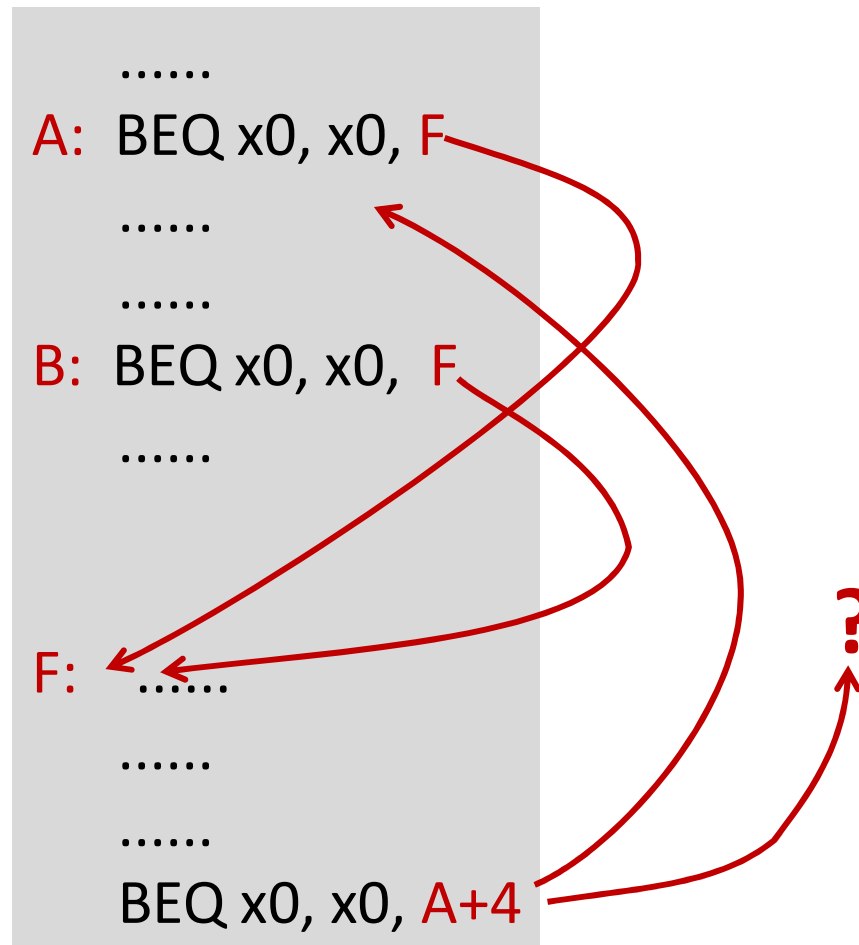
```
for (int* a=&A[0]; a<&A[16]; a++) {
    sum+=*a;
}
```

- Assembly Code (suppose `sum`, `A`, `a` are in r_{sum} , r_A , r_a)

```

    addi  r_a  r_A  0           # a=&A[0]
L1:  lw    r_tmp  0(r_a)        # sum+=*a
    add   r_sum r_sum r_tmp
    addi  r_a  r_a  4           # a++
    addi  r_tmp r_A  64         # tmp=&A[16]
    bltu  r_a  r_tmp  L1
```


Function Call and Return



A function return need to

1. jump back to different callers
2. know where to jump back to

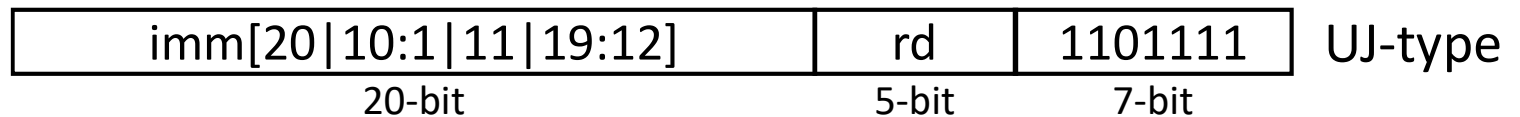
Jump and Link Instruction

- Assembly

JAL *rd* *imm*₂₁

Note: implicit *imm*[0]=0

- Machine encoding



- Semantics

– target = PC + sign-extend(*imm*₂₁)

– GPR[*rd*] ← PC + 4

– PC ← target

How far can you jump?

- Exceptions: misaligned target (4-byte)

Jump Indirect Instruction

- Assembly

JALR *rd*, *rs1*, *imm*₁₂

- Machine encoding

imm[11:0]	rs1	000	rd	1100111
12-bit	5-bit	3-bit	5-bit	7-bit

- Semantics

- $\text{target} = \text{GPR}[\text{rs1}] + \text{sign-extend}(\text{imm}_{12})$
- $\text{target} \&= 0\text{xffff_fffe}$
- $\text{GPR}[\text{rd}] \leftarrow \text{PC} + 4$
- $\text{PC} \leftarrow \text{target}$

How far can you jump?

- Exceptions: misaligned target (4-byte)

Assembly Programming 401

Caller

```
... code A ...
JAL x1, _myfxn
... code C ...
JAL x1, _myfxn
... code D ...
```

Callee

```
_myfxn:    ... code B ...
           JALR x0, x1, 0
```

- **A** $\rightarrow_{\text{call}}$ **B** $\rightarrow_{\text{return}}$ **C** $\rightarrow_{\text{call}}$ **B** $\rightarrow_{\text{return}}$ **D**
- How do you pass argument between caller and callee?
- If **A** set **x10** to 1, what is the value of **x10** when **B** returns to **C**?
- What registers can **B** use?
- What happens to **x1** if **B** calls another function

Caller and Callee Saved Registers

- Callee-Saved Registers
 - caller says to callee, “The values of these registers should not change when you return to me.”
 - callee says, “If I need to use these registers, I promise to save the old values to memory first and restore them before I return to you.”
- Caller-Saved Registers
 - caller says to callee, “If there is anything I care about in these registers, I already saved it myself.”
 - callee says to caller, “Don’t count on them staying the same values after I am done.”
- Unlike endianness, this is not arbitrary

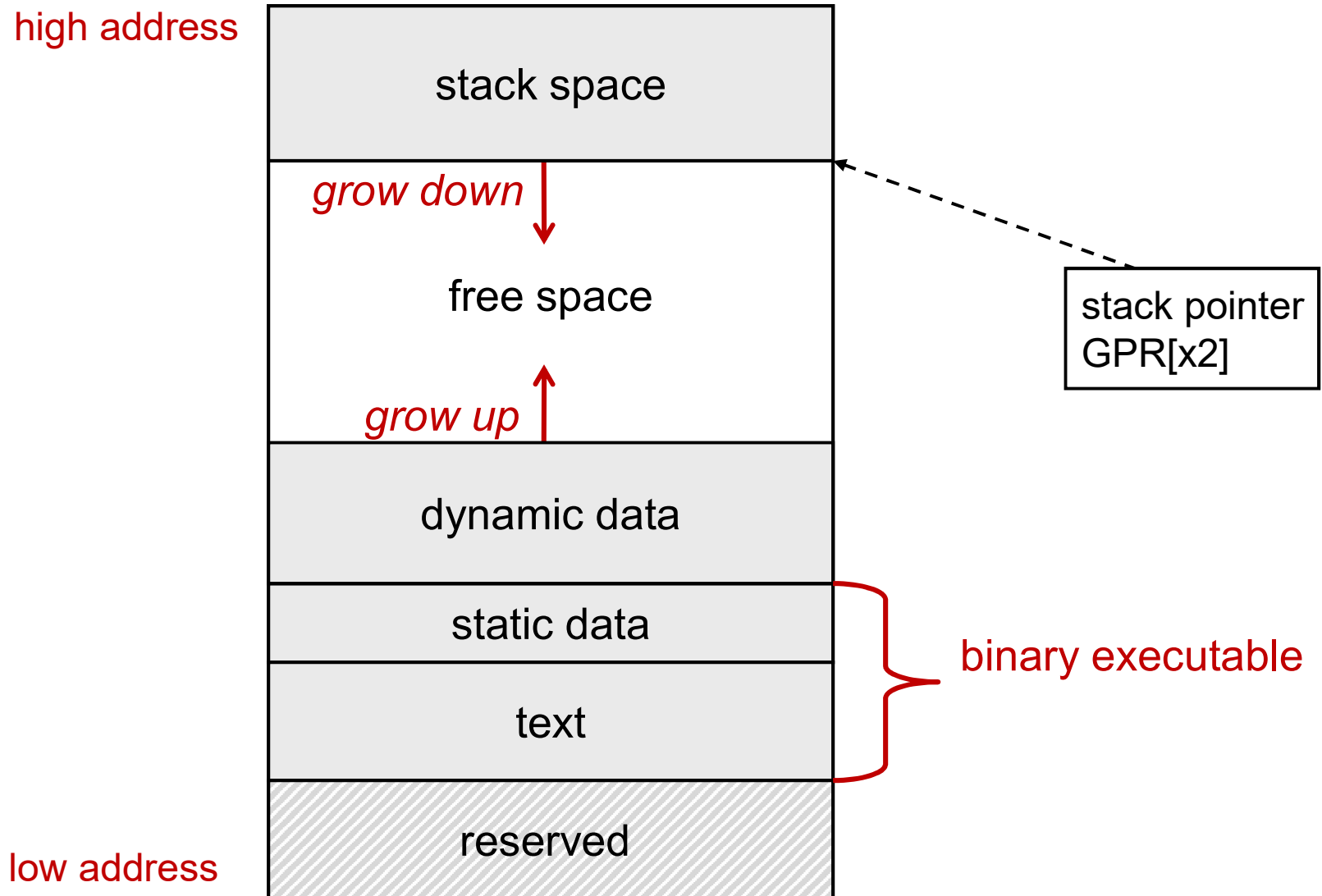
When to use which?

RISC-V Register Usage Convention

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

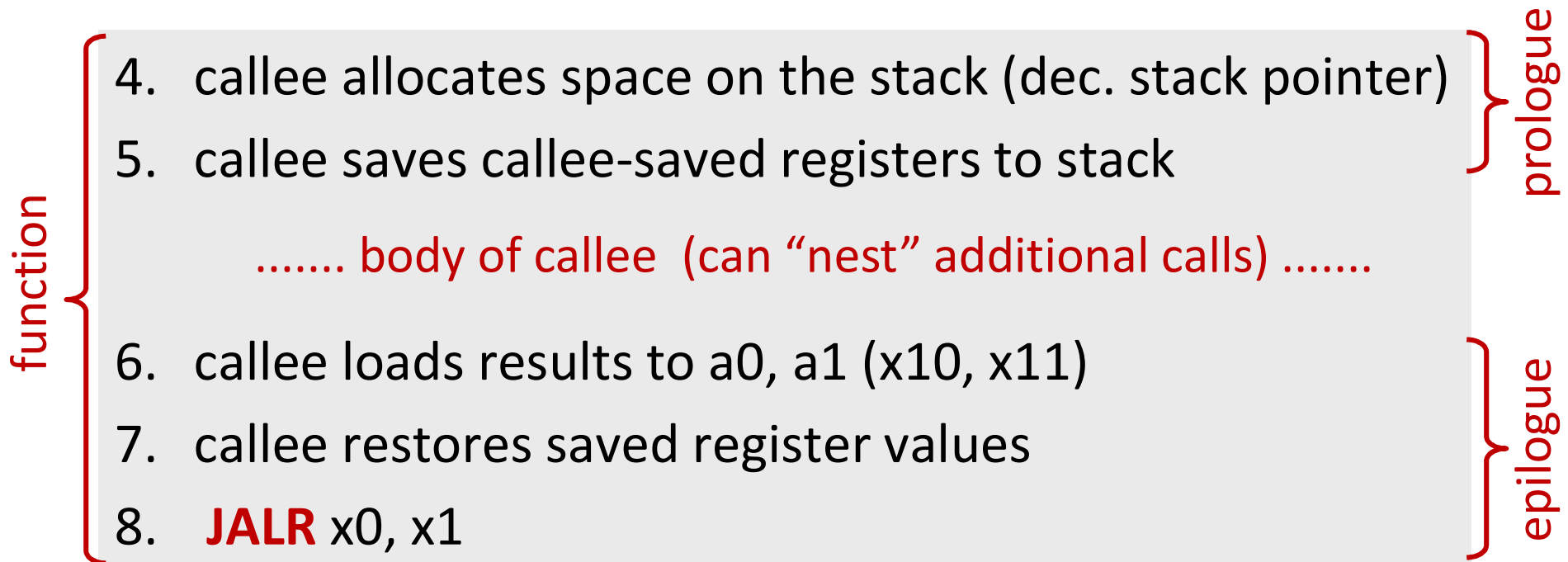
[The RISC-V Instruction Set Manual]

Memory Usage Convention



Basic Calling Convention

1. caller saves caller-saved registers
2. caller loads arguments into a0~a7 (x10~x17)
3. caller jumps to callee using **JAL** x1



9. caller continues with return values in a0, a1

Terminologies

- Instruction Set Architecture
 - machine state and functionality as observable and controllable by the programmer
- Instruction Set
 - set of commands supported
- Machine Code
 - instructions encoded in binary format
 - directly consumable by the hardware
- Assembly Code
 - instructions in “textual” form, e.g. `add r1, r2, r3`
 - converted to machine code by an assembler
 - one-to-one correspondence with machine code
(mostly true: compound instructions, labels)

We didn't talk about

- Privileged Modes
 - user vs. supervisor
- Exception Handling
 - trap to supervisor handling routine and back
- Virtual Memory
 - each process has 4-GBytes of private, large, linear and fast memory?
- Floating-Point Instructions