18-447 Lecture 2:
RISC-V Instruction Set Architecture

James C. Hoe

Department of ECE

Carnegie Mellon University

Housekeeping

- Your goal today
  - get bootstrapped on RISC-V RV32I to start Lab 1 (will revisit general ISA issues in L4)
- Notices
  - Student survey on Canvas, due next Wed
  - H02: Lab 1, Part A, due week of 1/27
  - H03: Lab 1, Part B, due week of 2/3
- Readings
  - P&H Ch2
  - P&H Ch4.1~4.4 (next time)

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

- Microarchitecture Level

  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

- Realization Level

  machined alloy gears vs stamped sheet metal

  CMOS vs ECL vs vacuum tubes

conceptual

physical

[Computer Architecture, Blaauw and Brooks, 1997]
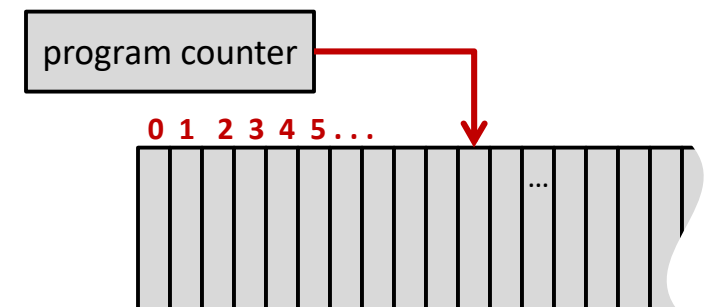
Stored Program Architecture
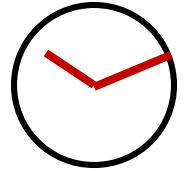## a.k.a. von Neumann

- 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 invention

program counter

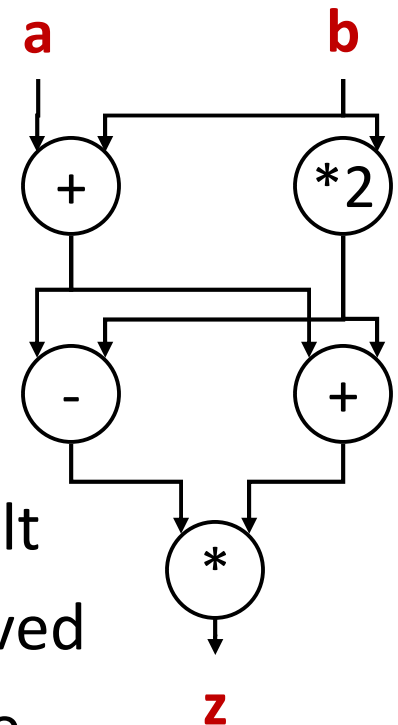0 1 2 3 4 5 . . .

...

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?

```
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]

Parallel Random Access Memory

$$P_1 \quad P_2 \quad P_3 \quad \bullet \bullet \bullet \bullet \bullet \quad P_n$$

Memory
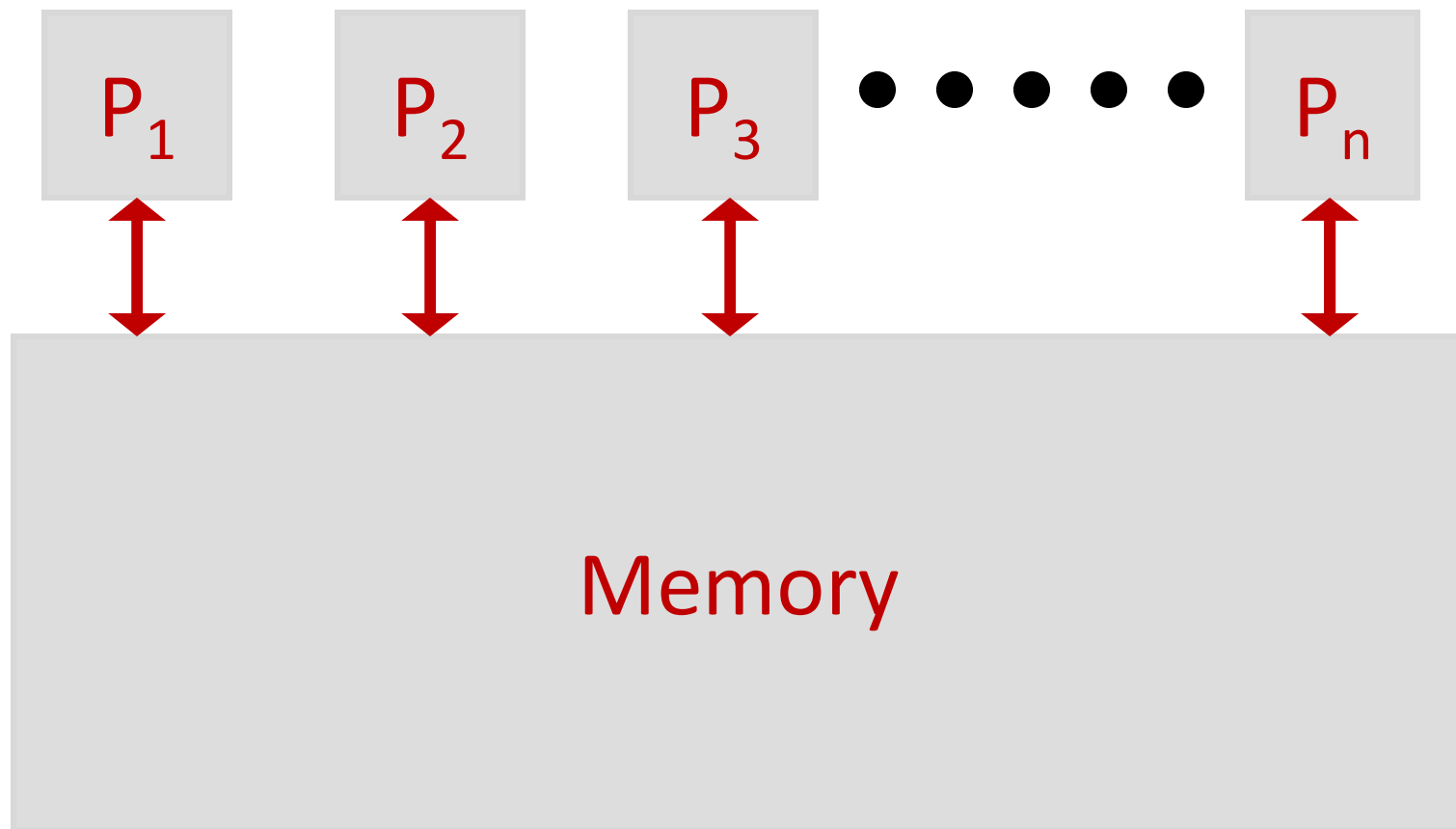
Do you naturally think parallel or sequential?

Instruction Set Architecture (ISA)

"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 ← target address
    - else PC ← 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

note x0=0
x1
x2
general purpose register file
32x 32-bit words named x0...x31

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

32-bit memory 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

    – GPR[rd] ← GPR[rs1] + GPR[rs2]

    – PC ← 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}

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-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

[from page 54, The RISC-V Instruction Set Manual]

Reg-Immediate ALU Instructions

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

    ADDI rd, rs1, imm$_{12}$

- Machine encoding

imm[11:0]	rs1	000	rd	0010011
12-bit	5-bit	3-bit	5-bit	7-bit

- Semantics
    - GPR[rd] ← GPR[rs1] + sign-extend (imm)
    - PC ← 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 R-type encoding   32-bit I-type ALU

Note: SLTIU does <u>unsigned</u> compare with <u>sign</u>-extended immediate

[from page 54, 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.)
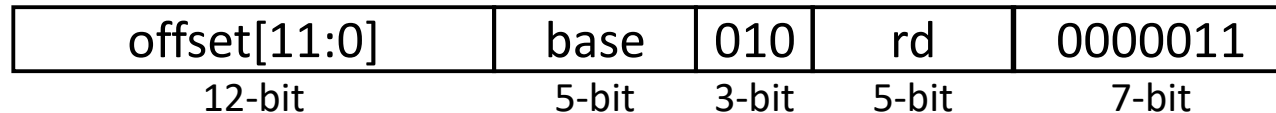 - does not effect memory or other registers

Load Instructions

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

  LW rd, offset$_{12}$(base) ← rs1

- Machine encoding

offset[11:0]	base	010	rd	0000011
12-bit	5-bit	3-bit	5-bit	7-bit

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

- Exceptions: none for now

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

  e.g., LB ::   GPR[rd] ← sign-extend(MEM$_8$[byte_address])

  LBU :: GPR[rd] ← zero-extend(MEM$_8$[byte_address])

  Note: RV32I memory is byte-addressable, little-endian

Big Endian vs. Little Endian

(Part I, Chapter 4, Gulliver's Travels)

- 32-bit signed or unsigned integer word is 4 bytes

- By convention we "write" MSB on left

MSB
(most significant)

8-bit	8-bit	8-bit	8-bit

LSB
(least significant)
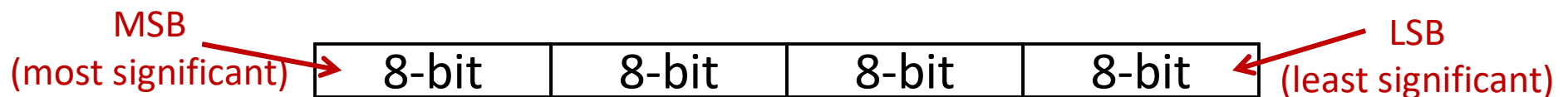
- On a byte-addressable machine . . . .

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	

- Common case is aligned loads and stores
  - 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
  - byte_address$_{32}$ = sign-extend(offset$_{12}$) + GPR[base]
  - MEM$_{32}$[byte_address] ← GPR[rs2]
  - PC ← PC + 4

- Exceptions: none for now

- Variations: SW, SH, SB

  e.g., SB::   MEM$_8$[byte_address] ← (GPR[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  r_temp  0(r_A)        # r_temp = A[0]
  add r_temp  r_h  r_temp   # r_temp = h + A[0]
  SW  r_temp  32(r_A)       # A[8] = r_temp
                            # 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

[from page 54, The RISC-V Instruction Set Manual]

RV32I Immediate Encoding

- RV32I adopts 2 different register-immediate formats (I vs S) to keep rs2 operand at inst[24:20] always

- Most RISCs had 1 register-immediate format

opcode	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

  – rt field used as a source (e.g., store) or dest (e.g., load)

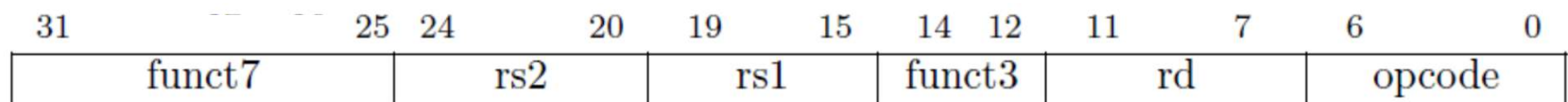 – also common to opt for longer 16-bit immediate

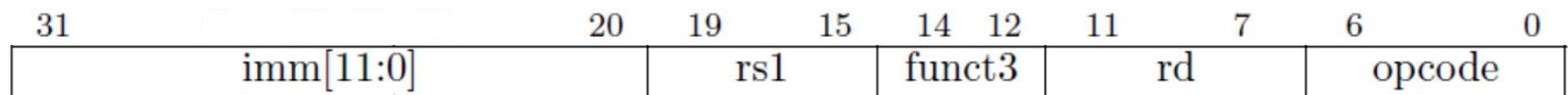- RV32I encodes immediate in non-consecutive bits
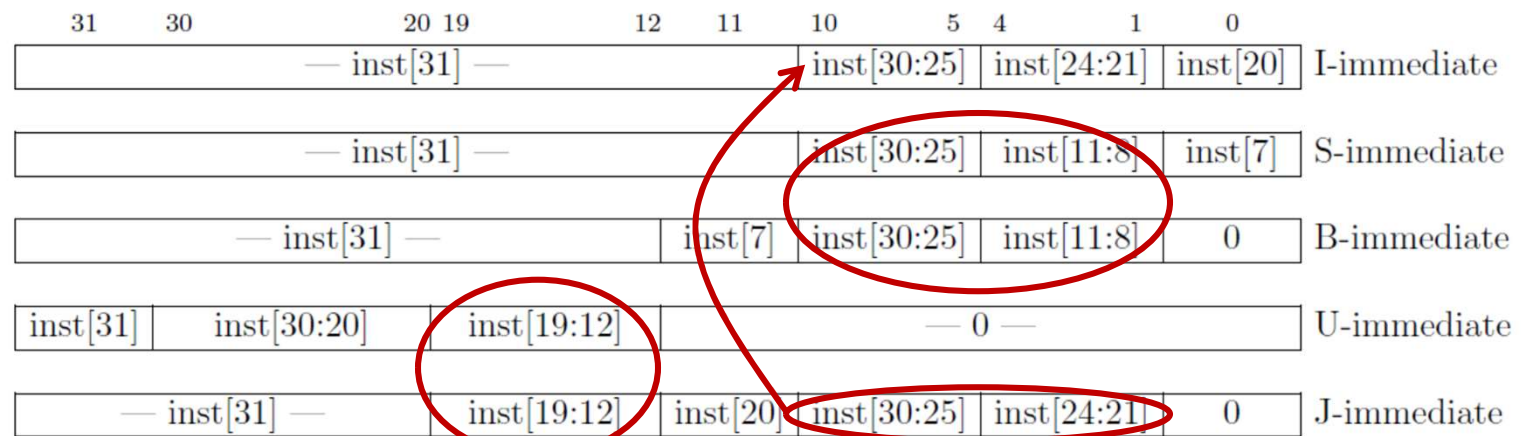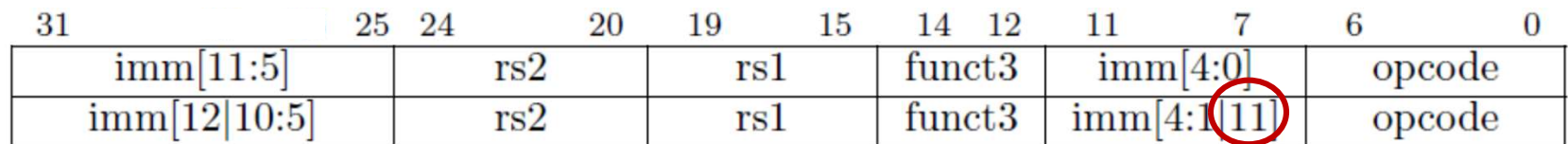
RV32I Instruction Formats

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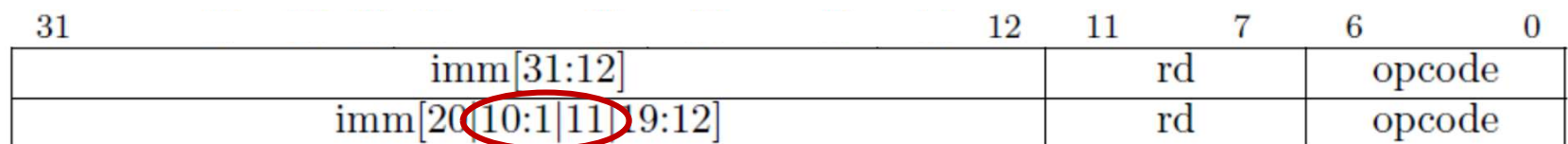
- R-type: 3 register operands

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode

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

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode

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

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode

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

31 12	11 7	6 0
imm[31:12]	rd	opcode
imm[20|10:1|11|19:12]	rd	opcode

Aimed to simplify decoding and field extraction

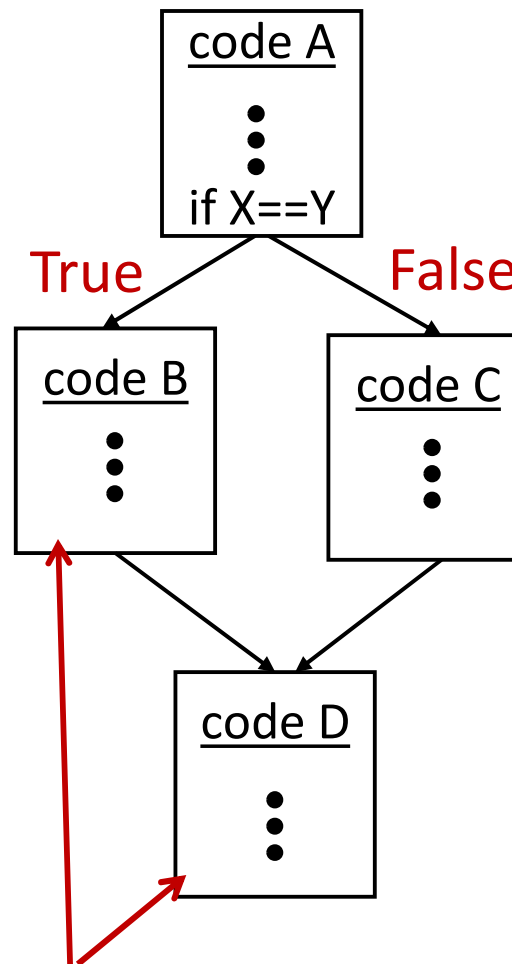# Control Flow Instructions

- C-Code

{ code A }

if X==Y then

    { code B }

else

    { code C }

{ code D }

Control Flow Graph

code A

⋮

if X==Y

True   False

code B   code C

⋮   ⋮

code D

⋮

Assembly Code
(linearized)

code A

⋮

if X==Y
goto

code C

⋮

goto

code B

⋮

code D

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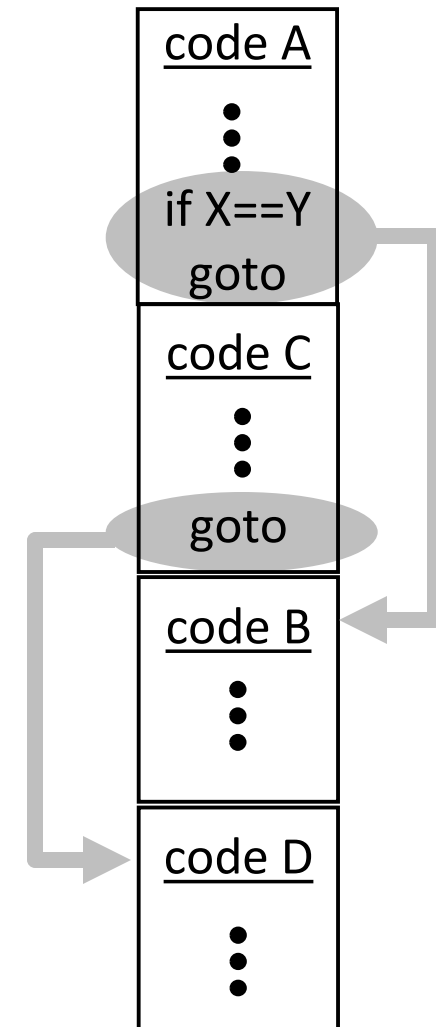
basic blocks (1-way in, 1-way out, all or nothing)

(Conditional) Branch Instructions

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

    BEQ rs1, rs2, imm$_{13}$ Note: implicit imm[0]=0

- Machine encoding

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011
7-bit	5-bit	5-bit	3-bit	5-bit	7-bit

- Semantics
  - target = PC + sign-extend(imm$_{13}$)
  - if GPR[rs1]==GPR[rs2]   then   PC $\leftarrow$ target
                                 else   PC $\leftarrow$ PC + 4

                                 How far can you jump?
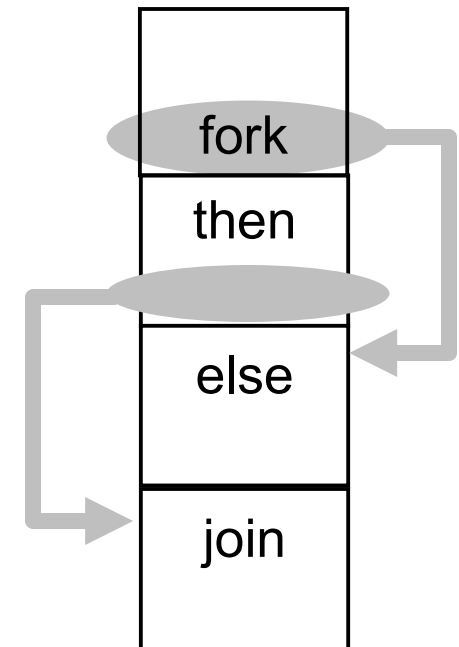
- Exceptions: misaligned target (4-byte) if taken
- Variations
  - BEQ, BNE, BLT, BGE, BLTU, BGEU

Assembly Programming 301

- E.g. High-level Code

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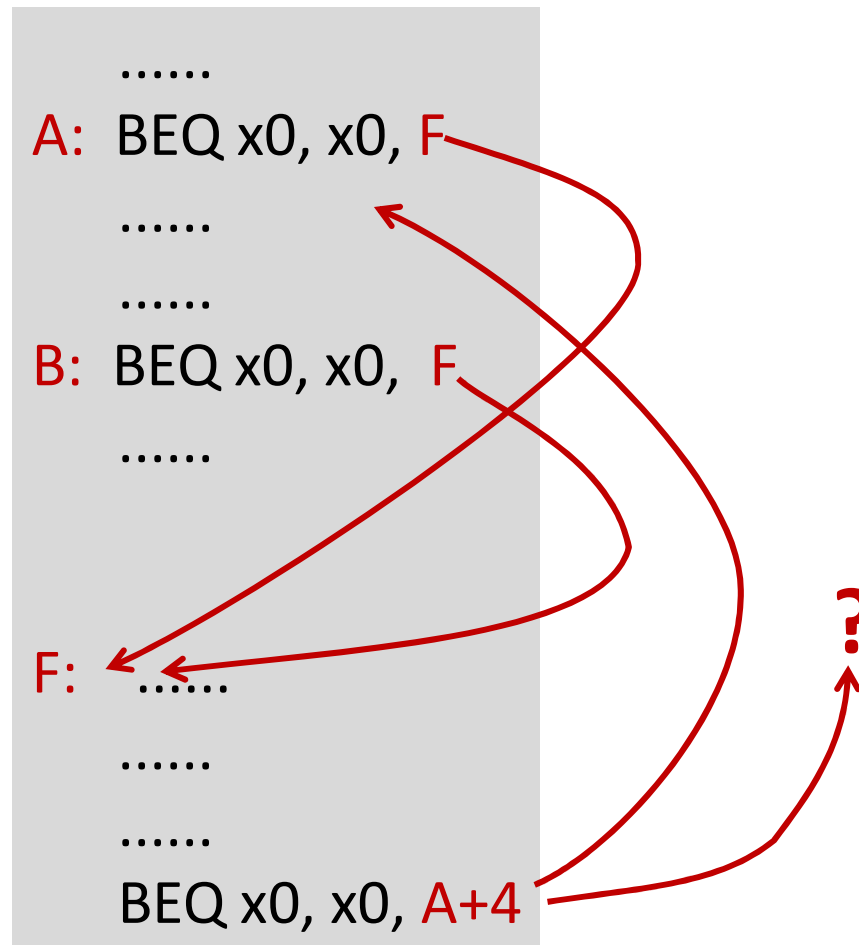

fork

then

else

join

- Assembly Code

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

```
        bne r_i r_j L1      # L1 and L2 are addr labels
                            # assembler computes offset
        add r_e r_g x0      # e = g
        beq x0 L2
L1:     add r_e r_h x0      # e = h
L2:     add r_f r_e x0      # f = e
```

Function Call and Return

A function return need to 1. jump back to different callers
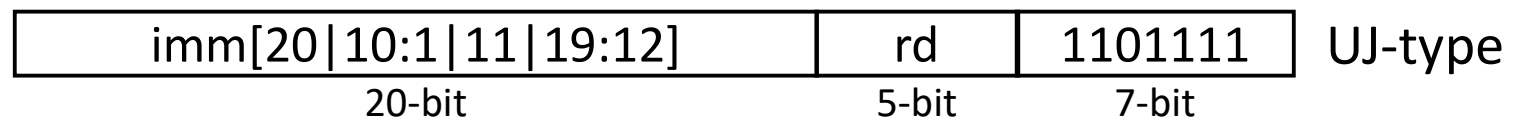   2. know where to jump back to

Jump and Link Instruction

- Assembly

  JAL rd imm$_{21}$ Note: implicit imm[0]=0

- Machine encoding

imm[20|10:1|11|19:12]	rd	1101111	UJ-type
20-bit	5-bit	7-bit	

- 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

  – target = GPR[rs1] + sign-extend(imm_{12})

  – target &= 0xffff_fffe

  – GPR[rd] ← PC + 4

  – PC ← 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** →$_{call}$ **B** →$_{return}$ **C** →$_{call}$ **B** →$_{return}$ **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.
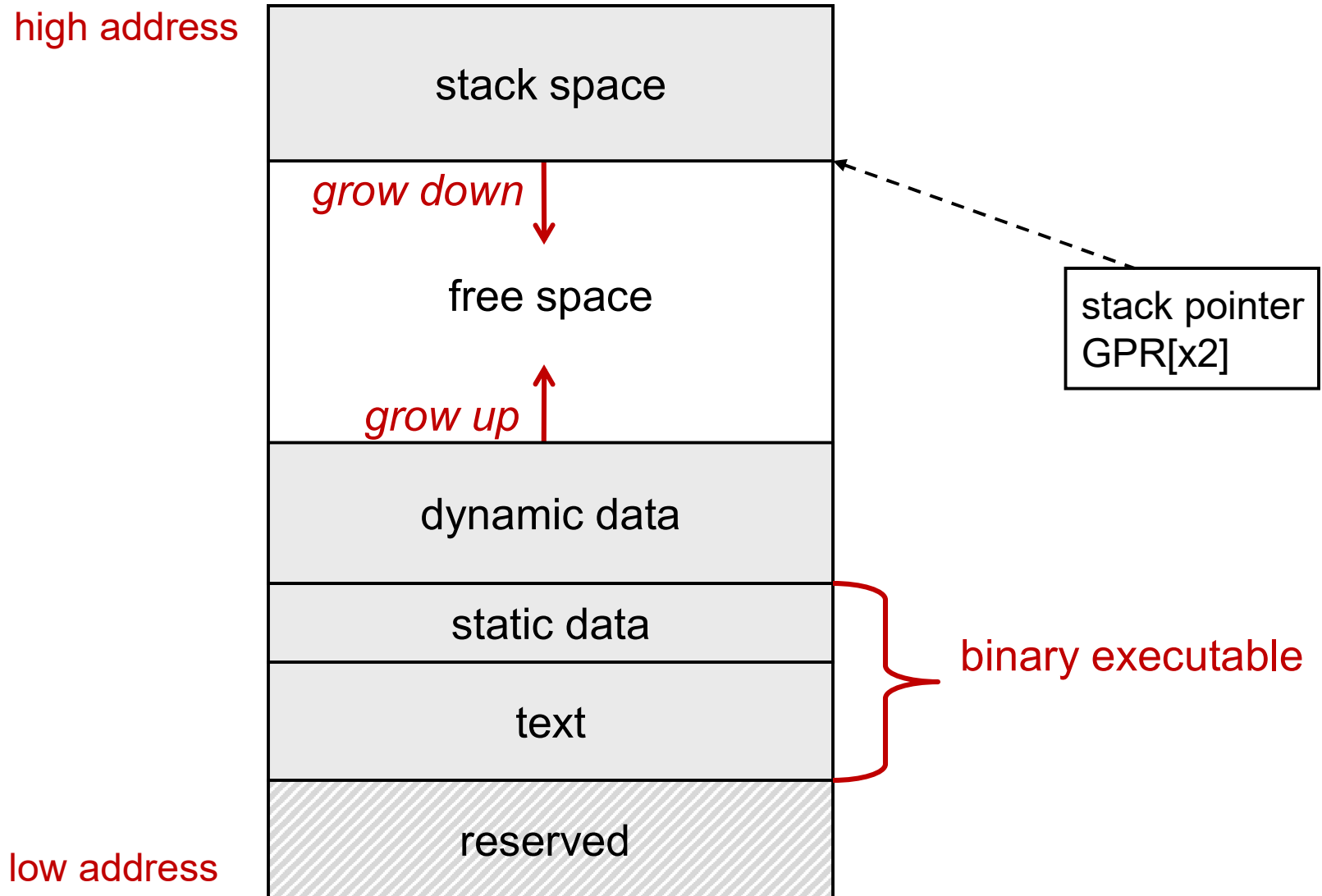- Unlike endianness, this is not arbitrary

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

[from page 100, The RISC-V Instruction Set Manual]

Memory Usage Convention

high address

stack space

grow down

free space

grow up

dynamic data

static data

text

reserved

low address

stack pointer
GPR[x2]

binary executable

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

4. callee allocates space on the stack (dec. stack pointer)

5. callee saves callee-saved registers to stack

.... body of callee (can "nest" additional calls) ....

6. callee loads results to a0, a1 (x10, x11)

7. callee restores saved register values

8. **JALR** x0, x1
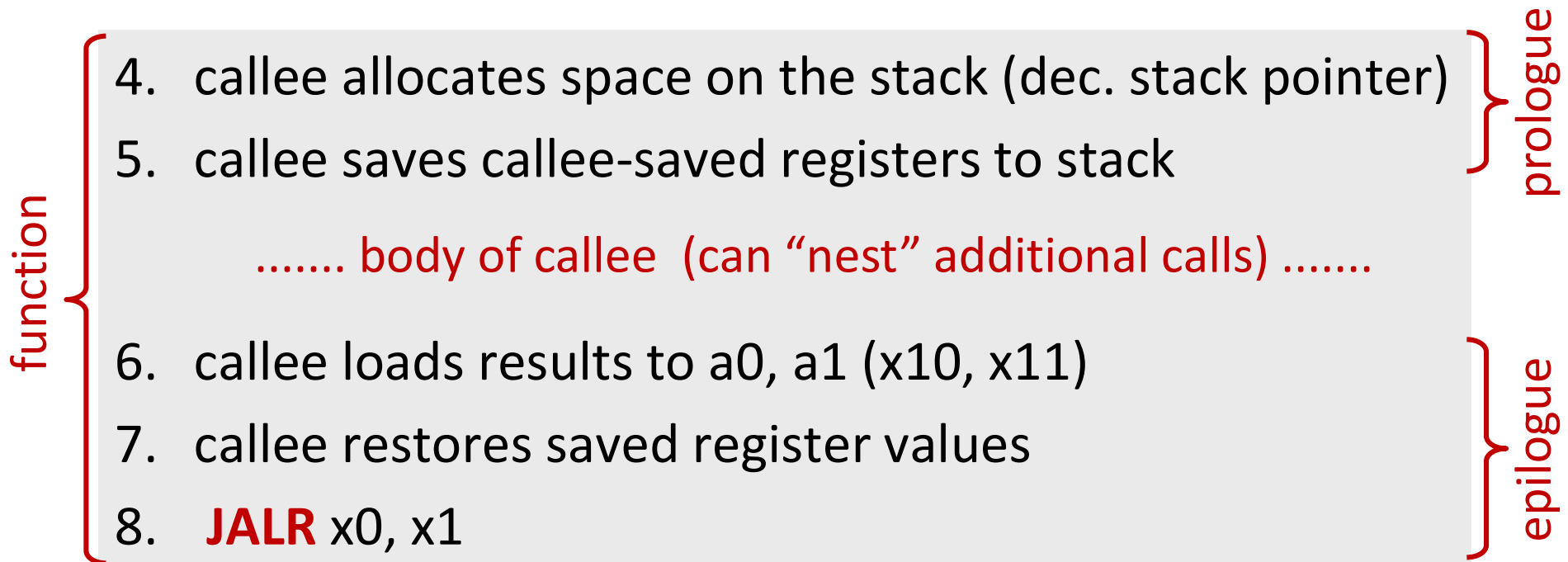
9. caller continues with return values in a0, a1

prologue

epilogue

function

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