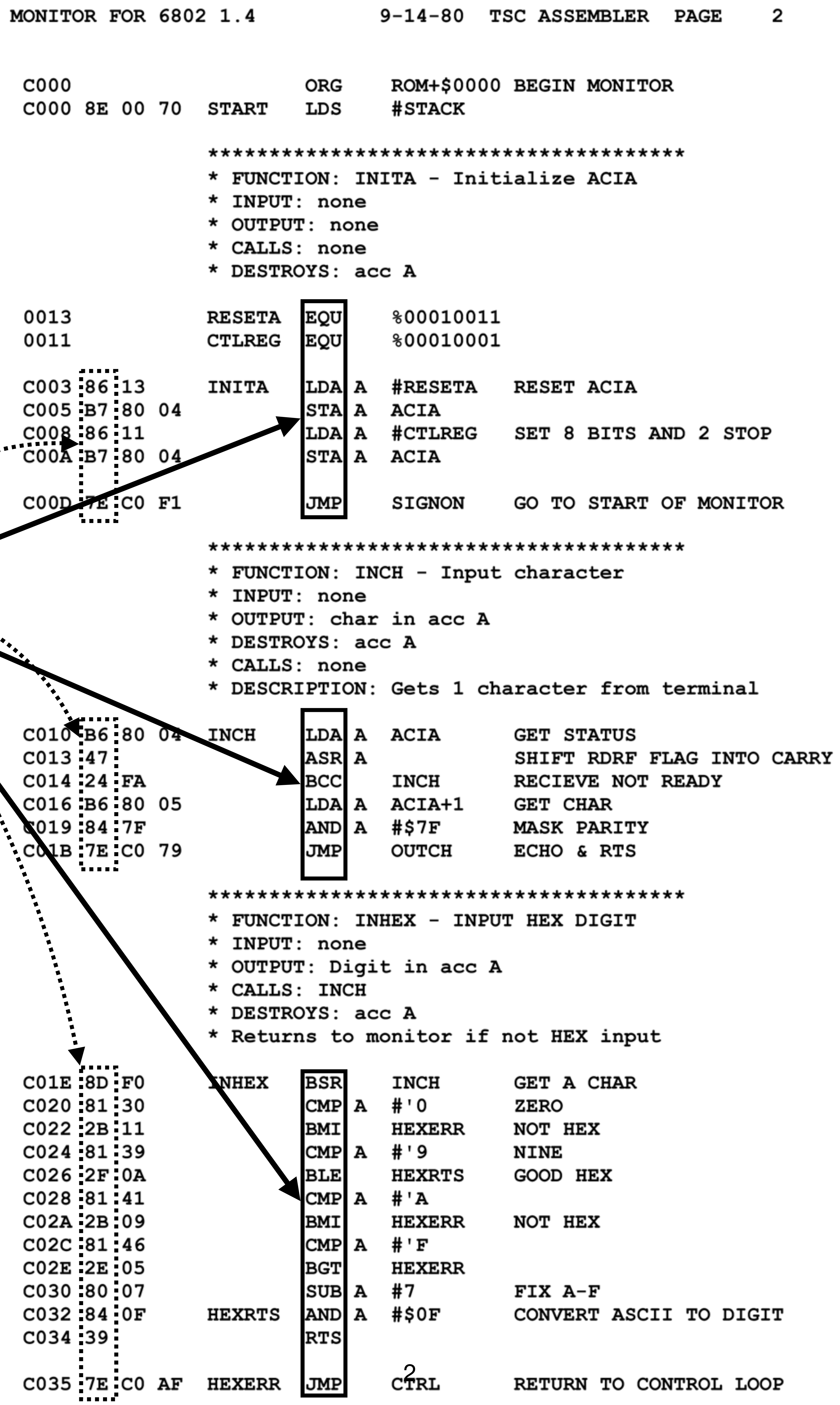
Assembly for Architecture

Lab Tonight! 7-8:15pm
Edmunds 105

Assembly program for the Motorola MC6800 microprocessor (1974)

197 “opcodes”



```
MONITOR FOR 6802 1.4         9-14-80  TSC ASSEMBLER  PAGE    2

 C000                ORG    ROM+$0000 BEGIN MONITOR
 C000 8E 00 70  START   LDS    #STACK

               ***************************************
               * FUNCTION: INITA - Initialize ACIA
               * INPUT: none
               * OUTPUT: none
               * CALLS: none
               * DESTROYS: acc A

 0013           RESETA  EQU    %00010011
 0011           CTLREG  EQU    %00010001

 C003 86 13     INITA   LDA A  #RESETA   RESET ACIA
 C005 B7 80 04          STA A  ACIA
 C008 86 11             LDA A  #CTLREG   SET 8 BITS AND 2 STOP
 C00A B7 80 04          STA A  ACIA

 C00D 7E C0 F1          JMP    SIGNON    GO TO START OF MONITOR

               ***************************************
               * FUNCTION: INCH - Input character
               * INPUT: none
               * OUTPUT: char in acc A
               * DESTROYS: acc A
               * CALLS: none
               * DESCRIPTION: Gets 1 character from terminal

 C010 B6 80 04  INCH    LDA A  ACIA      GET STATUS
 C013 47                ASR A            SHIFT RDRF FLAG INTO CARRY
 C014 24 FA             BCC    INCH      RECIEVE NOT READY
 C016 B6 80 05          LDA A  ACIA+1    GET CHAR
 C019 84 7F             AND A  #$7F      MASK PARITY
 C01B 7E C0 79          JMP    OUTCH     ECHO & RTS

               ***************************************
               * FUNCTION: INHEX - INPUT HEX DIGIT
               * INPUT: none
               * OUTPUT: Digit in acc A
               * CALLS: INCH
               * DESTROYS: acc A
               * Returns to monitor if not HEX input

 C01E 8D F0     INHEX   BSR    INCH      GET A CHAR
 C020 81 30             CMP A  #'0       ZERO
 C022 2B 11             BMI    HEXERR    NOT HEX
 C024 81 39             CMP A  #'9       NINE
 C026 2F 0A             BLE    HEXRTS    GOOD HEX
 C028 81 41             CMP A  #'A
 C02A 2B 09             BMI    HEXERR    NOT HEX
 C02C 81 46             CMP A  #'F
 C02E 2E 05             BGT    HEXERR
 C030 80 07             SUB A  #7        FIX A-F
 C032 84 0F     HEXRTS  AND A  #$0F      CONVERT ASCII TO DIGIT
 C034 39                RTS

 C035 7E C0 AF  HEXERR  JMP    CTRL      RETURN TO CONTROL LOOP
```

Image Credit: Michael Holley, https://en.wikipedia.org/wiki/Assembly_language

Outline

- Interpreting and constructing an assembly instruction
- Assembly instruction formats
- Assembly instruction classes
- Assembly language trade-offs

I apologize in advance, the first part of today is a lot for me speaking at you… There will be reinforcing exercises and please ask questions as they come up!

Memory

13	03	c3	fe	00	00	00	ef	be	ad	de	ee	ff	c0
0xff00	0xff01	0xff02	0xff03	0xff04	0xff05	0xff06	0xff07	0xff08	0xff09	0xff0a	0xff0b	0xff0c	0xff0d

What is 4-byte instruction at address `0xff00`?

How should the processor interpret `0xfec30313`?

Interpreting an Instruction

- Instructions are composed of an *operation* segment (sometimes referred as "*opcode*")
- Remaining fields in the instruction are defined by the operation type
- Depending on the instruction set, the remaining segments (*fields*) may or may not follow strict formats
- Register definitions require a translation to a byte-code interpretation

Sample assembly for reference

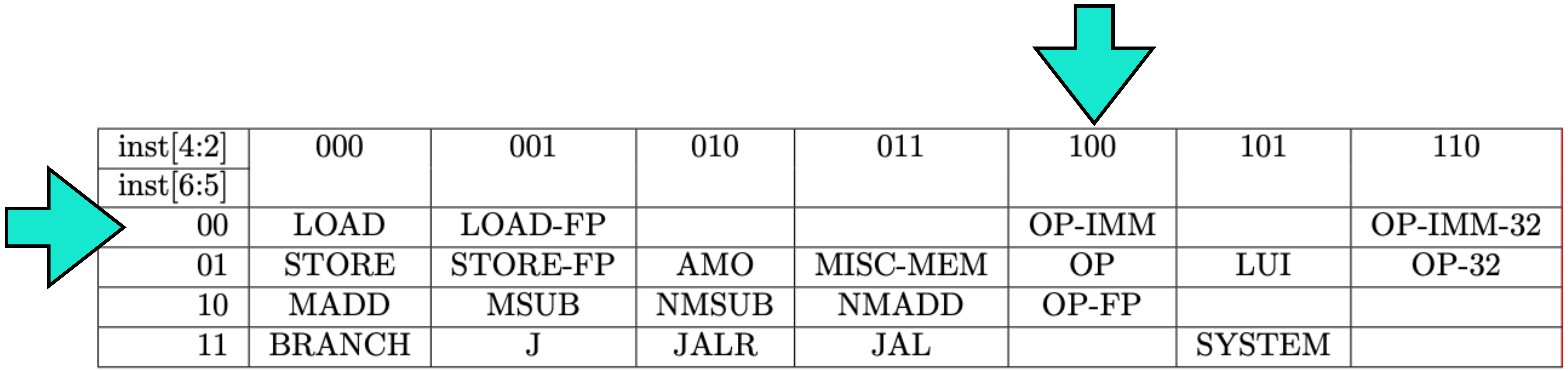
```
add   t0, t1, t2

addi  x4, x4, 15

bne   a1, a2, 0xffff
```

Interpreting a RISC-V Instruction

- RISC-V assembly instructions follow strict formats



inst[4:2] / inst[6:5]	000	001	010	011	100	101	110
00	LOAD	LOAD-FP			OP-IMM		OP-IMM-32
01	STORE	STORE-FP	AMO	MISC-MEM	OP	LUI	OP-32
10	MADD	MSUB	NMSUB	NMADD	OP-FP		
11	BRANCH	J	JALR	JAL		SYSTEM	

Table 2: RISC-V base opcode map, inst[1:0]=11

https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.pdf

$$\begin{tikzcd} \texttt{0xfec303}\boxed{\texttt{13}} \arrow[d] \\ \texttt{0b0}\boxed{\texttt{0010011}} \arrow[d] \\ \text{OP-IMM} \end{tikzcd}$$

Opcodes in Other ISAs

x86: http://ref.x86asm.net/coder32.html

ARM: https://iitd-plos.github.io/col718/ref/arm-instructionset.pdf

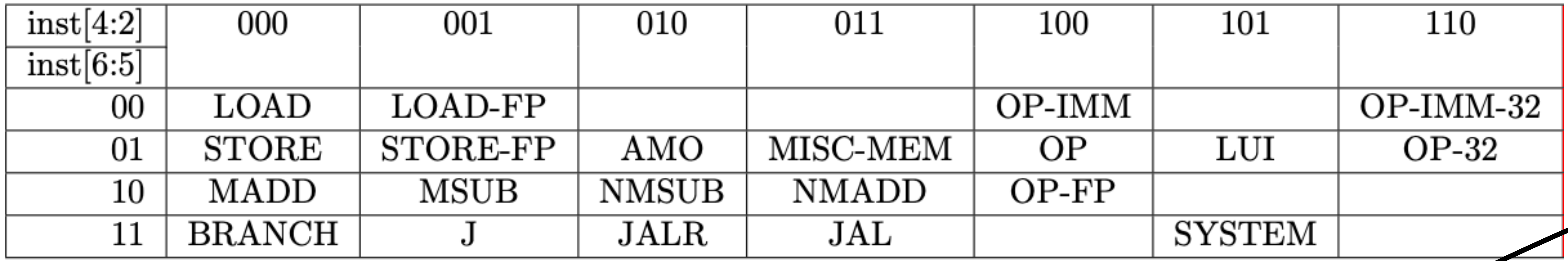
Power10: https://files.openpower.foundation/s/9izgC5Rogi5Ywmm

Assembly Instruction Formats

- Depending on the instruction set, different instructions may have different lengths
- When a processor wants to “fetch” an instruction to execute, it needs to know how many bytes to fetch from memory
- The processor can *over approximate* the size of the instruction to be the largest possible instruction size, and then use the relevant bits after the it is interpreted

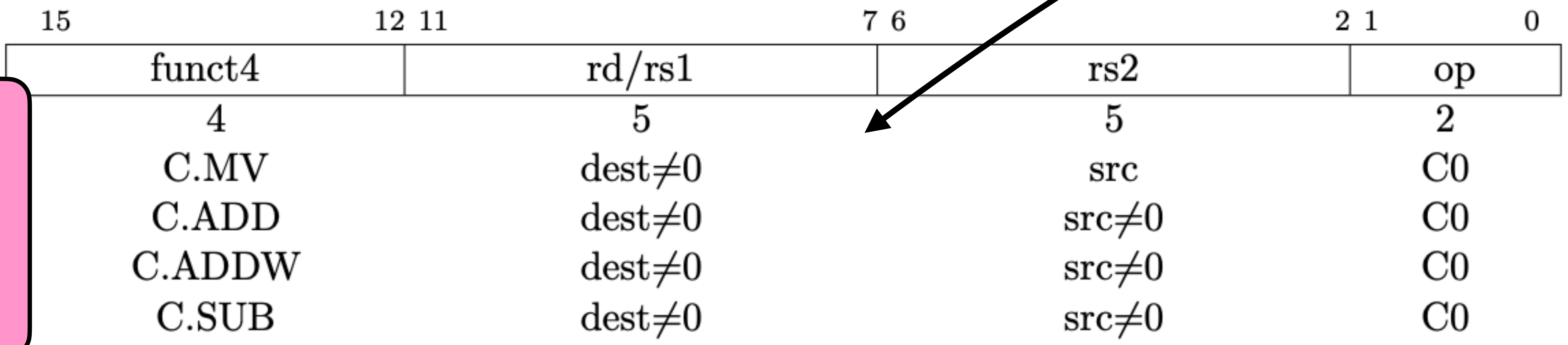
Assembly Instruction Formats (RISC-V Case Study)

https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.pdf



inst[4:2] / inst[6:5]	000	001	010	011	100	101	110
00	LOAD	LOAD-FP			OP-IMM		OP-IMM-32
01	STORE	STORE-FP	AMO	MISC-MEM	OP	LUI	OP-32
10	MADD	MSUB	NMSUB	NMADD	OP-FP		
11	BRANCH	J	JALR	JAL		SYSTEM	

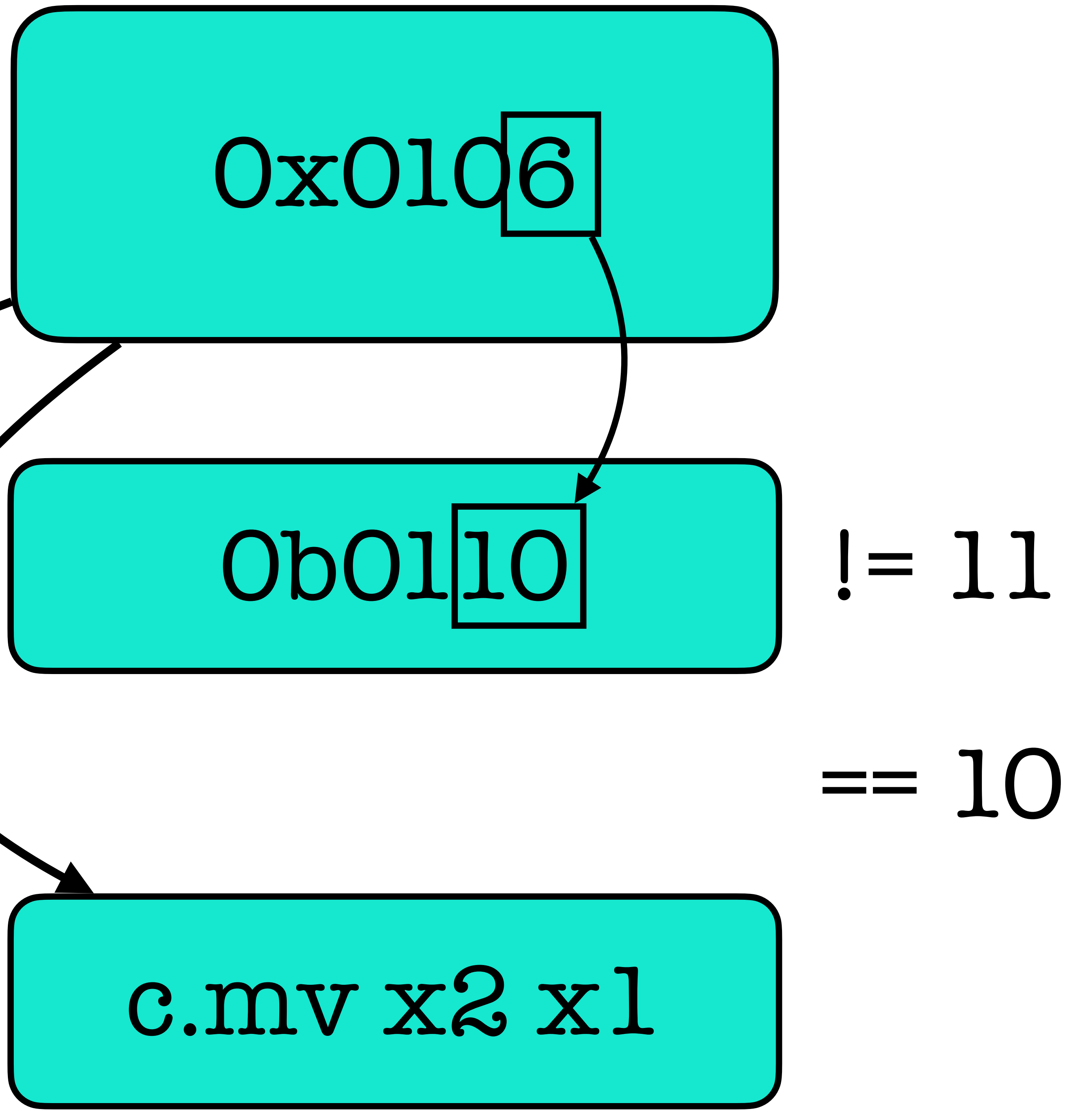
Table 2: RISC-V base opcode map, inst[1:0]=11



15–12	11–7	6–2	1–0
funct4	rd/rs1	rs2	op
4	5	5	2
C.MV	dest≠0	src	C0
C.ADD	dest≠0	src≠0	C0
C.ADDW	dest≠0	src≠0	C0
C.SUB	dest≠0	src≠0	C0

Compressed Instruction Format

https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2015-157.pdf

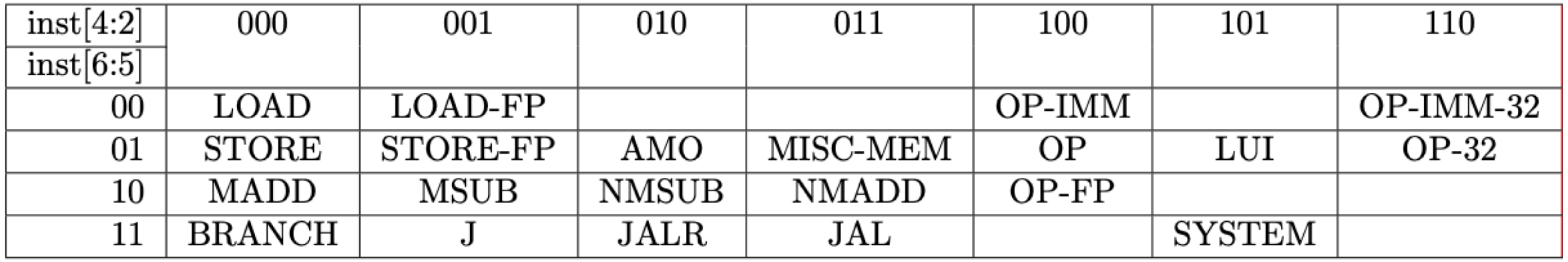


Assembly Instruction Formats (RISC-V Case Study)

- Instructions in the standard format are 32 bits long
- Instructions in the compressed format are 16 bits long
- All standard instructions have `0b11` as the least significant bits of the instruction because that is the opcode field

Chat with your neighbor(s)!

What is the operation for the RISC-V instruction "`0x3e820293`"?



inst[4:2] / inst[6:5]	000	001	010	011	100	101	110
00	LOAD	LOAD-FP			OP-IMM		OP-IMM-32
01	STORE	STORE-FP	AMO	MISC-MEM	OP	LUI	OP-32
10	MADD	MSUB	NMSUB	NMADD	OP-FP		
11	BRANCH	J	JALR	JAL		SYSTEM	

Table 2: RISC-V base opcode map, inst[1:0]=11

Chat with your neighbor(s)!

Why do you think the opcode field comprises the LSBs in RISC-V? How is this related to endianness?

Instruction Class Formats

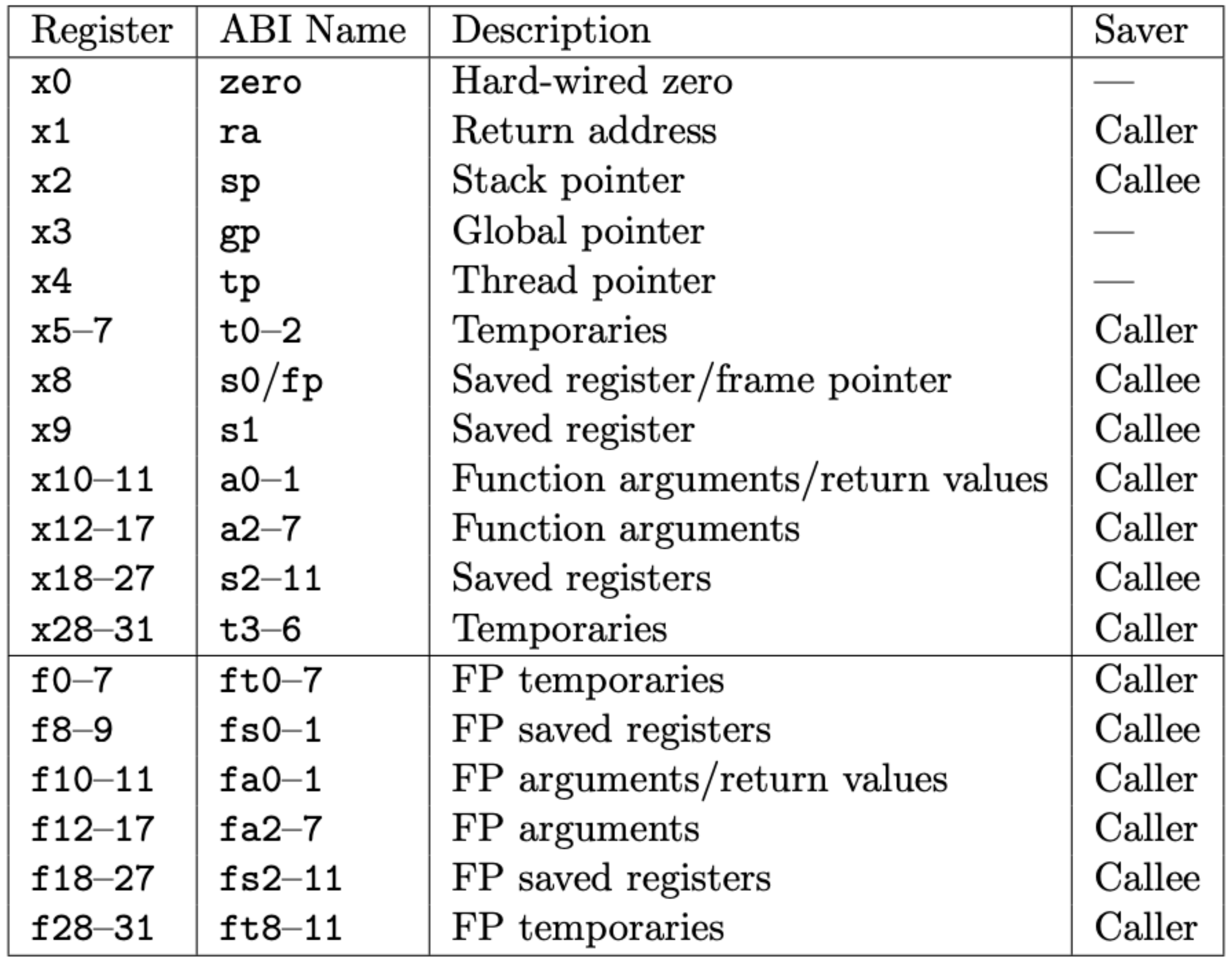
- Data transfers: loads and stores between memory and registers
- Control logic: conditional and unconditional jumps (support for if-statements, loops, function calls)
- Computations: add two variables together, subtract a constant, perform bitwise operations

Registers

- Small, fast memory (usually the size of a word, or default data size for a specific architecture)
- Used by the CPU
  - Usually addressed separately from main memory
  - Some have special purposes, some are general purpose (GPR)
- In RISC-V, all arithmetic and control computations are done on registers
  - To compute on memory, first load data from memory into register
  - Called a “register-register” architecture (other examples: ARM, MIPS)
  - Contrast with “register-memory” architecture (example: x86)

Registers (continued…)

- In RV32I: there are 31 GPRs (x1-x32) and each register has a conventional role
- x0 is hardwired to 0
- pc (program counter): holds the address of the current instruction



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
`f0–7`	`ft0–7`	FP temporaries	Caller
`f8–9`	`fs0–1`	FP saved registers	Callee
`f10–11`	`fa0–1`	FP arguments/return values	Caller
`f12–17`	`fa2–7`	FP arguments	Caller
`f18–27`	`fs2–11`	FP saved registers	Callee
`f28–31`	`ft8–11`	FP temporaries	Caller

Table 18.2: RISC-V calling convention register usage.

Chat with your neighbor(s)!

How many usable opcodes are in RISC-V assembly? What is advantageous about this encoding methodology? Problematic?

Takeaways

- Instructions are encoded with an “opcode” which tells a processor how to decode an instruction
- Different instruction types follow similar formats, and similar register definitions tend to appear in similar places
- Across instruction sets, various opcode conventions exist but in general decoding instructions generally starts by decoding opcodes