北京大学 PEKING UNIVERSITY

22530007

人工智能与芯片设计

3-单核CPU

燕博南

2023秋

Instruction Set Architecture (ISA)

- The contract between software and hardware
- Typically described by giving all the programmer-visible state (registers + memory) plus the semantics of the instructions that operate on that state
- IBM 360 was first line of machines to separate ISA from implementation (aka. *microarchitecture*)
- Many implementations possible for a given ISA
  - E.g., Soviets built code-compatible clones of the IBM360, as did Amdahl after he left IBM
  - E.g.2., AMD, Intel, VIA processors run the AMD64 ISA
  - E.g.3: many cellphones use the ARM ISA with implementations from many different companies including Apple, Qualcomm, Samsung, Huawei, etc.
- We use ARM/RISC-V as standard ISA in class (www.riscv.org)
  - Many companies and open-source projects build RISC-V implementations

ISA to Microarchitecture Mapping

- ISA often designed with particular microarchitectural style in mind, e.g.,

  Accumulator/Adder ⇒ hardwired, unpipelined
  CISC (Complex instruction set computer) ⇒ microcoded
  RISC (Reduced instruction set computer) ⇒ microcoded, pipelined
  VLIW (Very long instruction word) ⇒ fixed-latency in-order parallel pipelines
  JVM (Java virtual machine) ⇒ software interpretation

- But can be implemented with any microarchitectural style
  - Intel Ivy Bridge: hardwired pipelined CISC (x86) machine (with some microcode support)
  - Apple M1/M2 (native ARM ISA, emulates x86 in software)
  - ARM Jazelle: A hardware JVM processor
  - This lecture: a microcoded RISC-V machine

Control versus Datapath

- Processor designs can be split between
  - *datapath*, where numbers are stored and arithmetic operations computed, and
  - *control*, which sequences operations on datapath

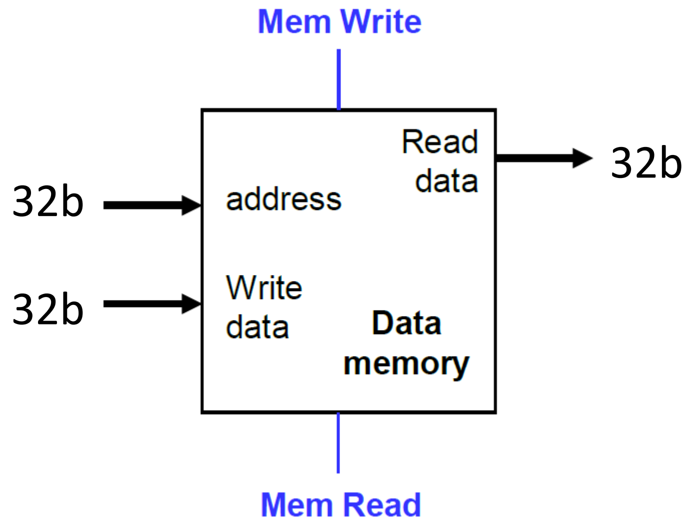
A computer is just a big fancy state machine.

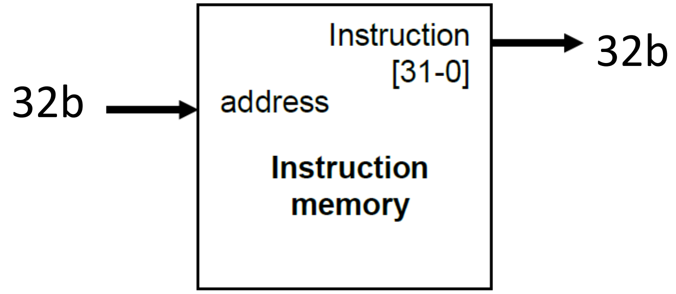
John von Neumann

- In the old days, “programming” involved actually changing a machine’s physical configuration by flipping switches or connecting wires.
  - A computer could run just one program at a time.
  - Memory only stored data that was being operated on.
- Then around 1944, John von Neumann and others got the idea to **encode instructions in a format that could be stored in memory just like data.**
  - The processor interprets and executes instructions from memory.
  - One machine could perform many different tasks, just by loading different programs into memory.
  - The“stored program” design is often called a Von Neumann machine.

Memories

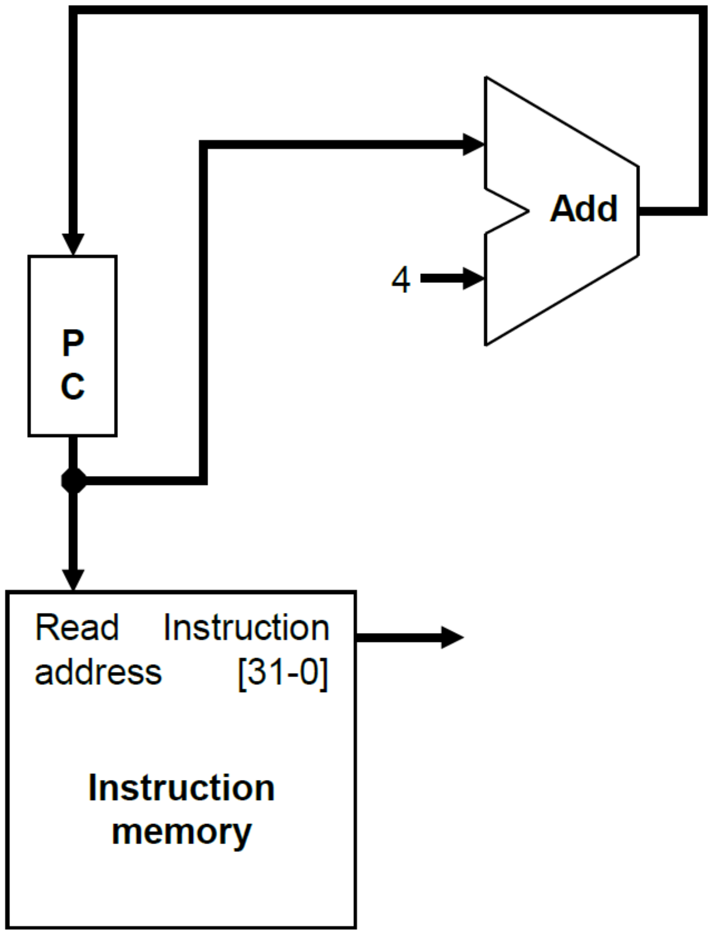
- Harvard architecture :
    - programs and data stored in separate memories.
- Blue lines represent control signals. MemRead and MemWrite should be set to 1 if the data memory is to be read or written respectively, and 0 otherwise.
- When a control signal does something when it is set to 1, we call it active high(vs. active low) because 1 is usually a higher voltage than 0.
- Pretend it’ s already loaded with a program, which doesn’ t change while it’ s running.





Instruction Fetching

- The CPU is in a infinite loop
- The program counter or PC register holds the address of the current instruction
- Given our instruction is 4 byte (32b) long
  - >> PC = PC + 4 after obtaining an instruction



Encoding R-type instructions

- Register-to-register arithmetic instructions use the R-type format.
  - op is the instruction opcode, and func specifies a particular arithmetic operation
  - rs, rt and rd are source and destination registers.

op	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Example

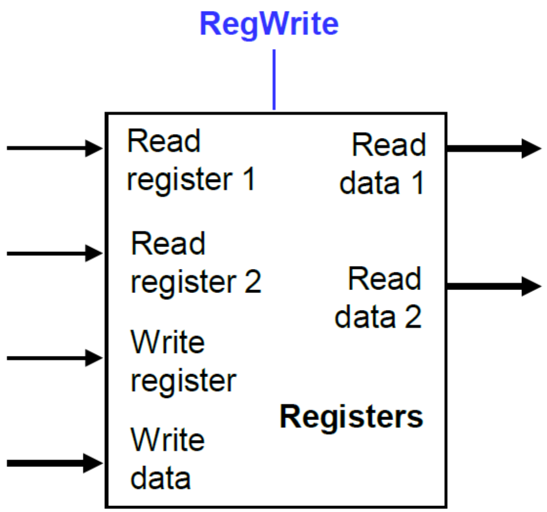
Now pretend you know assembly!

add $s4, $t1, $t2

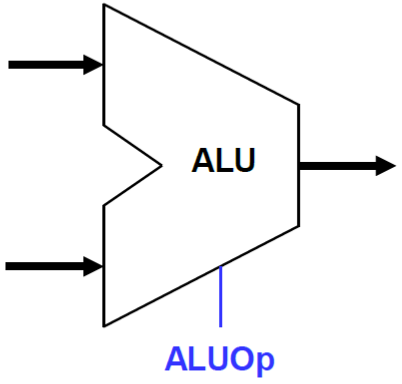
000000	01001	01010	10100	00000	1000000

Register File & ALU

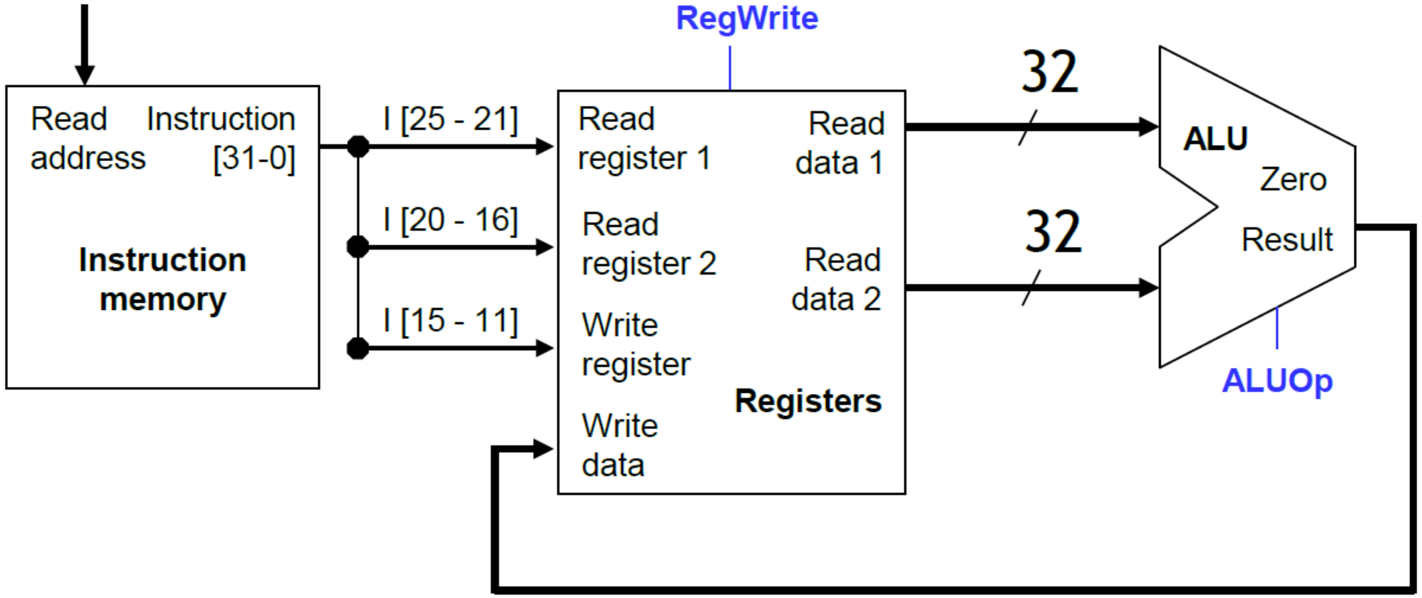
- R-type instructions must access registers and an ALU
- Our register file stores thirty-two 32-bit values.
  - Each register specifier is 5 bits long.
  - You can read from two registers at a time (2 ports).
  - RegWrite is 1 if a register should be written.
- Here's a simple ALU with five operations, selected by a 3-bit control signal ALUOp.



ALUOp	Function
000	and
001	or
010	add
110	subtract
111	slt



Executing an R-type instruction



- Fetch an instruction from “instruction memory”
- Fetch data from registers rs & rt
- ALU does computation
- Put results into rd

op	rs	rt	rd	shamt	func
31 26	25 21	20 16	15 11	10 6	5 0

Encoding I-type instructions

- Immediate number instructions (I-type)
  - Rt is the destination for lw, but a source for beq and sw
  - Address is a 16-bit signed constant

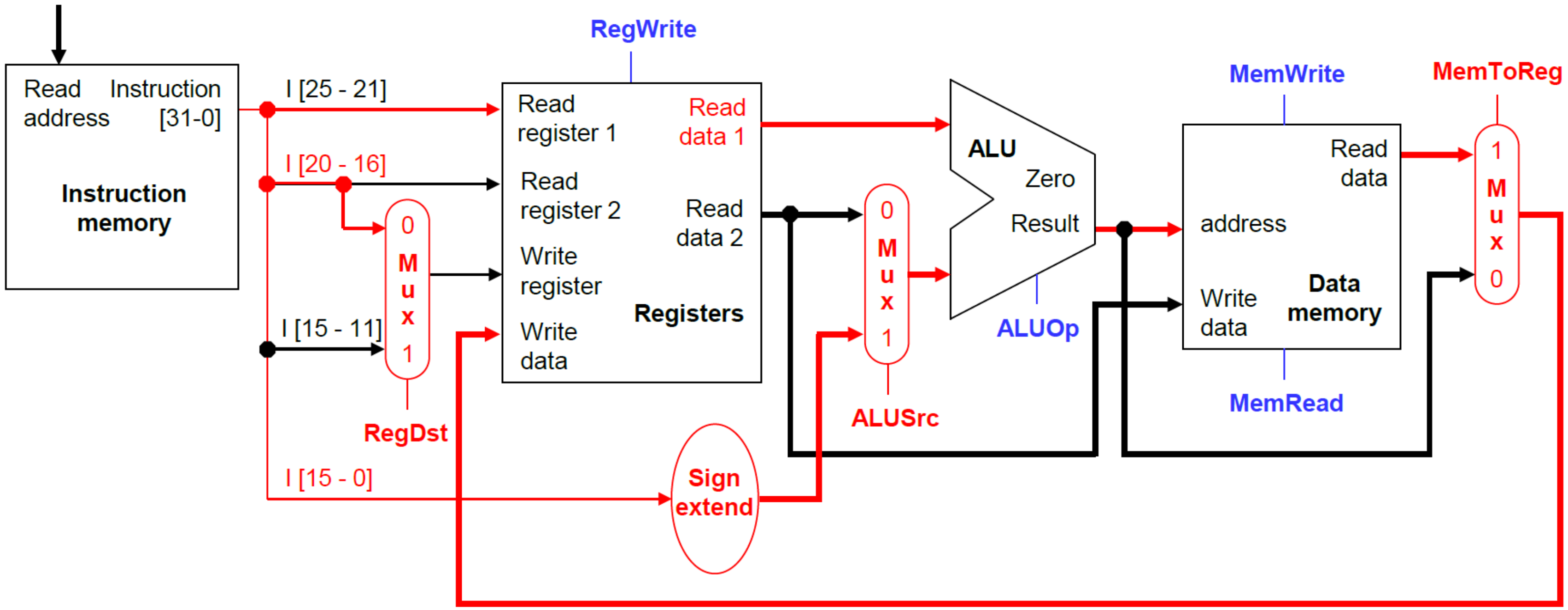
immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

- Example

```
ld x9, 64(x22) // Temporary reg x9 gets A[8]
```

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

Accessing Data Memory




```
ld x9, 64(x22) // Temporary reg x9 gets A[8]
```

Data memory Address: (the content in x22)+64

Operation: load the data in the "data memory" into x9

Branches

- For branch instructions, next PC should be obtained in the

```
        beq   $at, $0, L
        add   $v1, $v0, $0
        add   $v1, $v1, $v1
        j     somewhere
L:      add   $v1, $v0, $v0
```

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

BEQ: if rs1==rs2, then to go to the current PC+offset

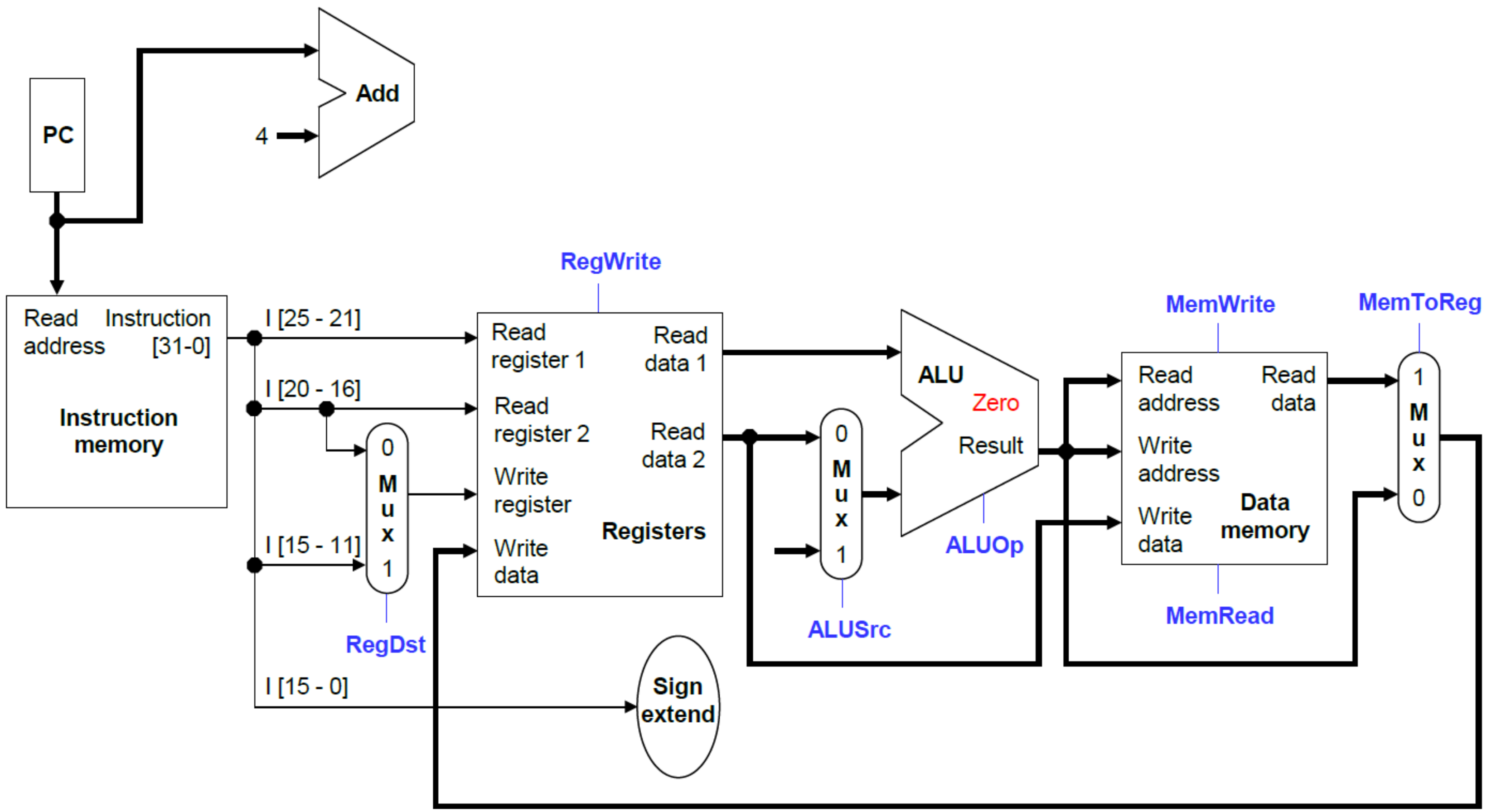
So Execute BEQ should be:

1. Fetch the instruction, like beq $at, $0, offset, from memory.

2. Compare $at and $0

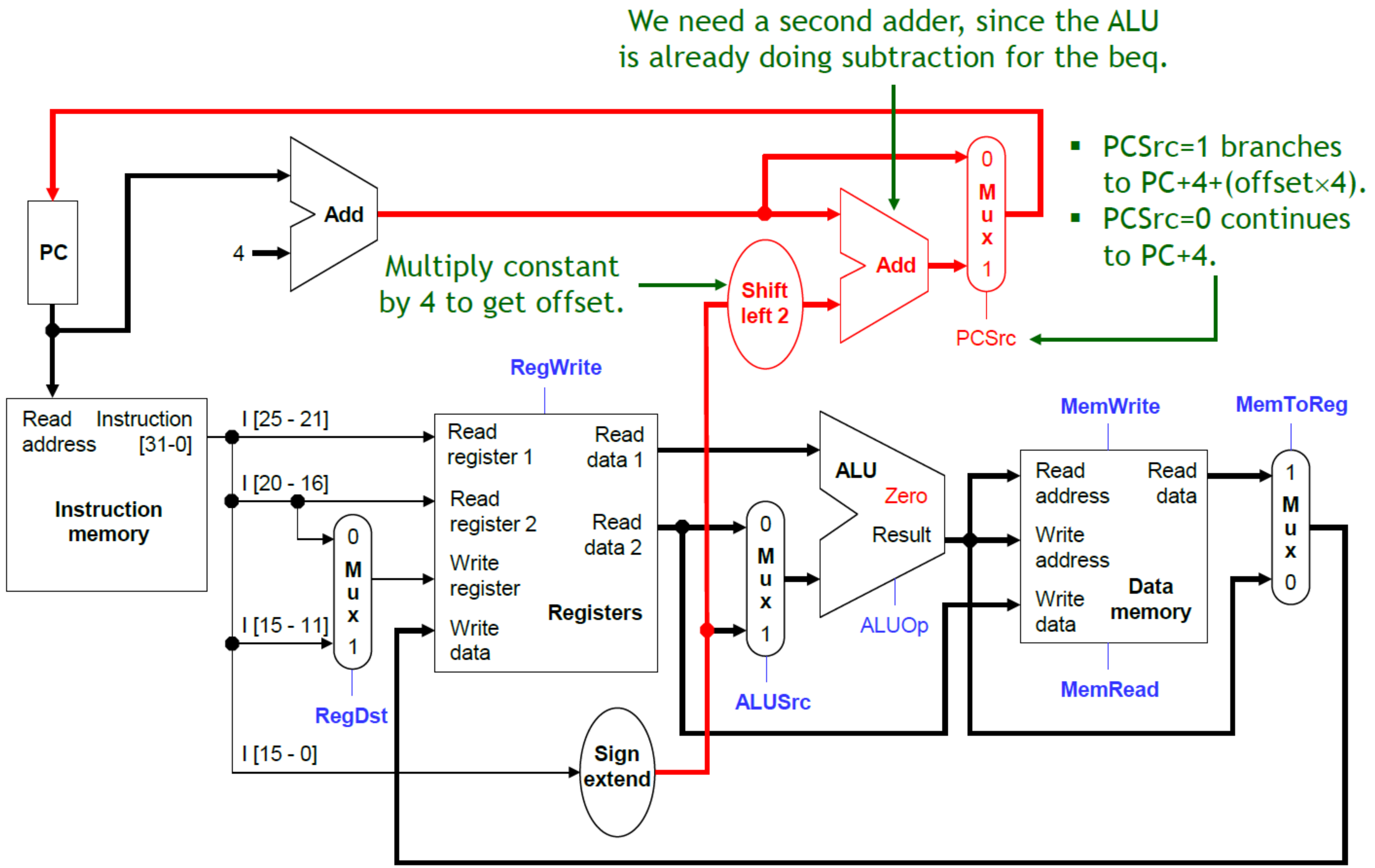
3. If yes, next PC = PC + offset * 4 byte/instruction

Hardware

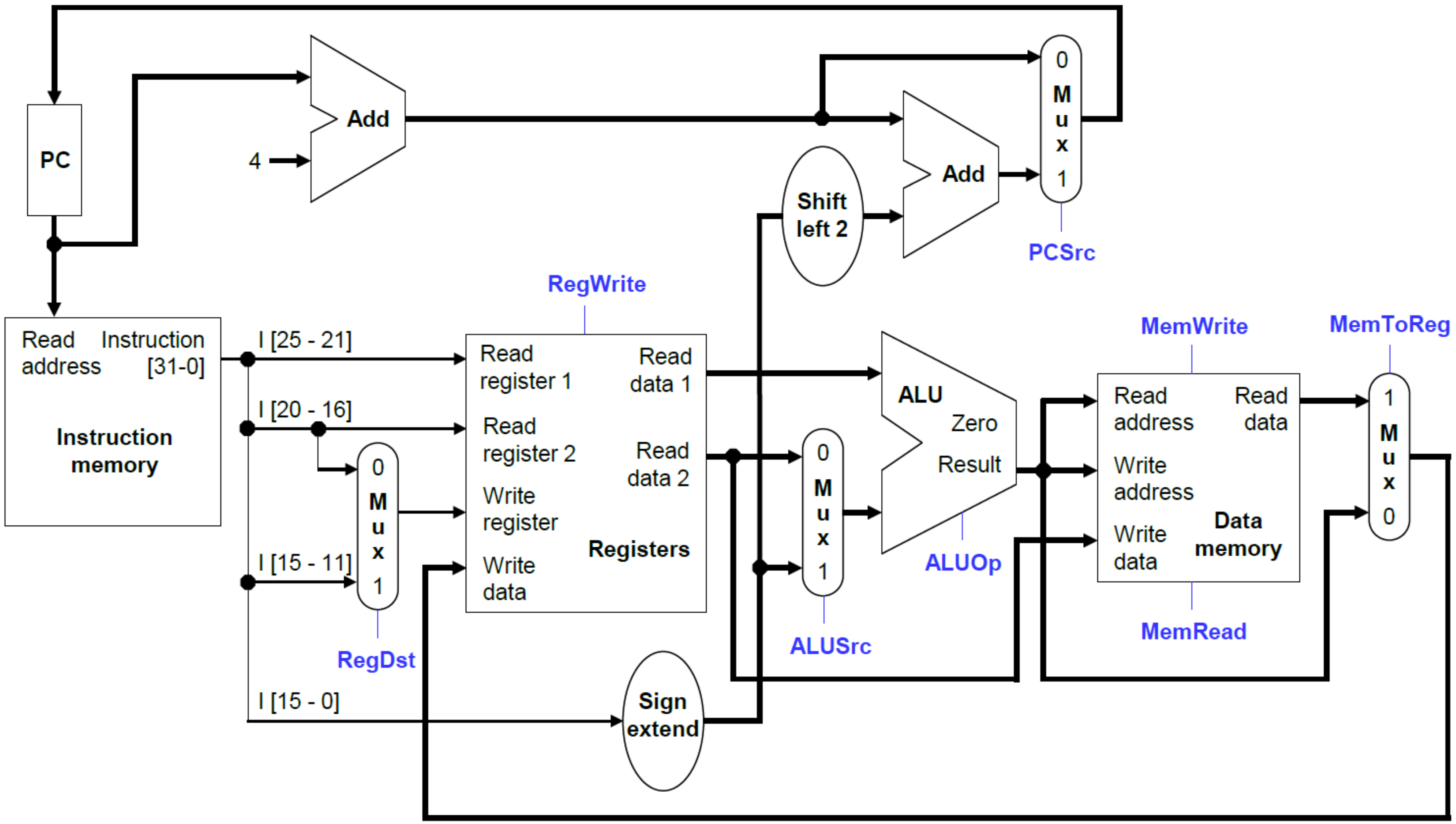


PC
Add
4
Read address
Instruction [31-0]
Instruction memory
I [25 - 21]
I [20 - 16]
I [15 - 11]
I [15 - 0]
0
M
u
x
1
RegDst
RegWrite
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Sign extend
0
M
u
x
1
ALUSrc
ALU
Zero
Result
ALUOp
MemWrite
Read address
Write address
Write data
Read data
Data memory
MemRead
MemToReg
1
M
u
x
0

Hardware


We need a second adder, since the ALU is already doing subtraction for the beq.
PCSrc=1 branches to PC+4+(offset×4).
PCSrc=0 continues to PC+4.
Multiply constant by 4 to get offset.
PC
Add
4
Shift left 2
Add
0
Mux
1
PCSrc
RegWrite
Read address
Instruction [31-0]
Instruction memory
I [25 - 21]
I [20 - 16]
I [15 - 11]
I [15 - 0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
0
Mux
1
RegDst
Sign extend
0
Mux
1
ALUSrc
ALU
Zero
Result
ALUOp
MemWrite
Read address
Write address
Write data
Read data
Data memory
MemRead
MemToReg
1
Mux
0

Final Hardware



Review A Little Bit

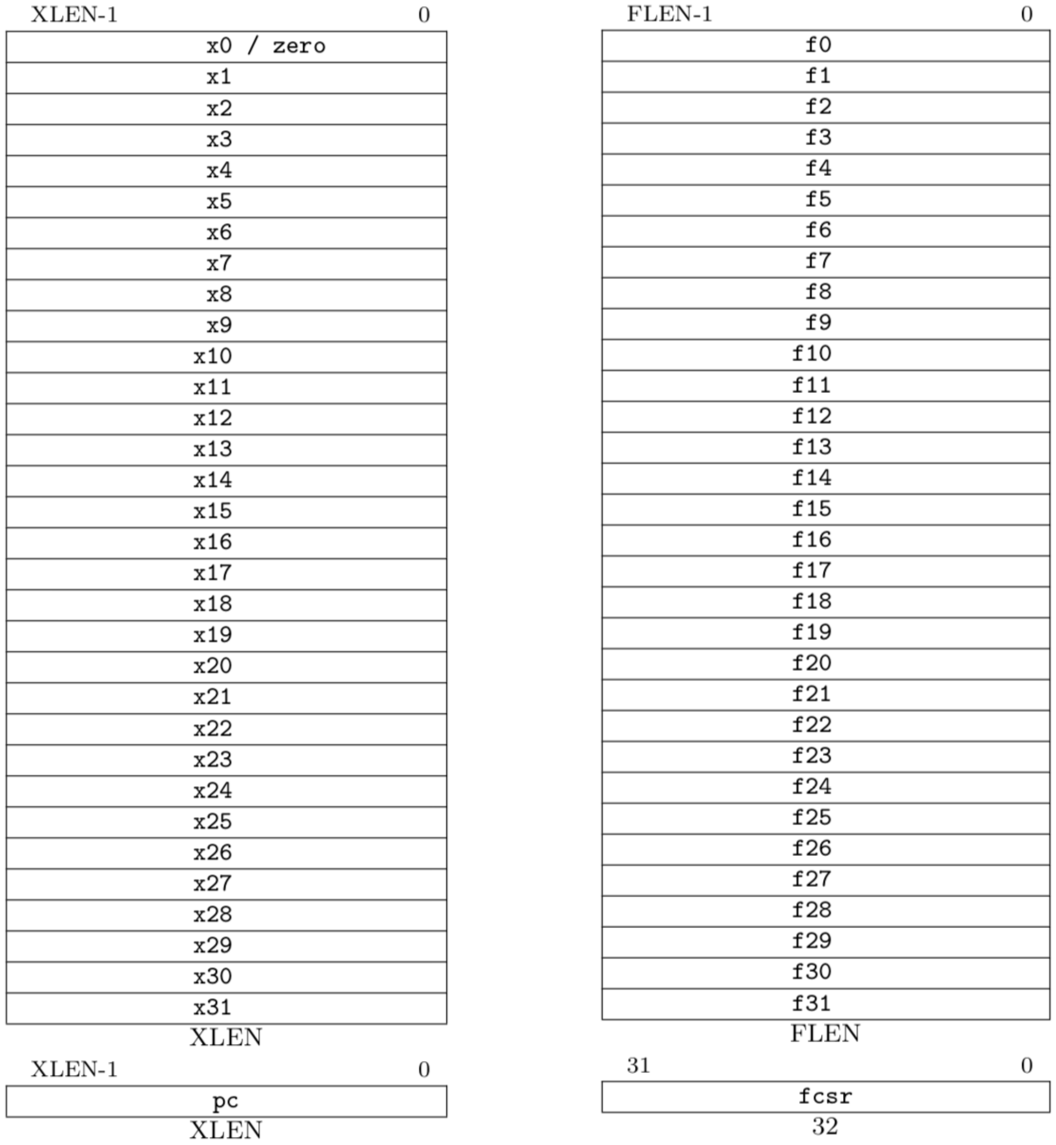
Review: RV32I Processor State

Program counter (**pc**)

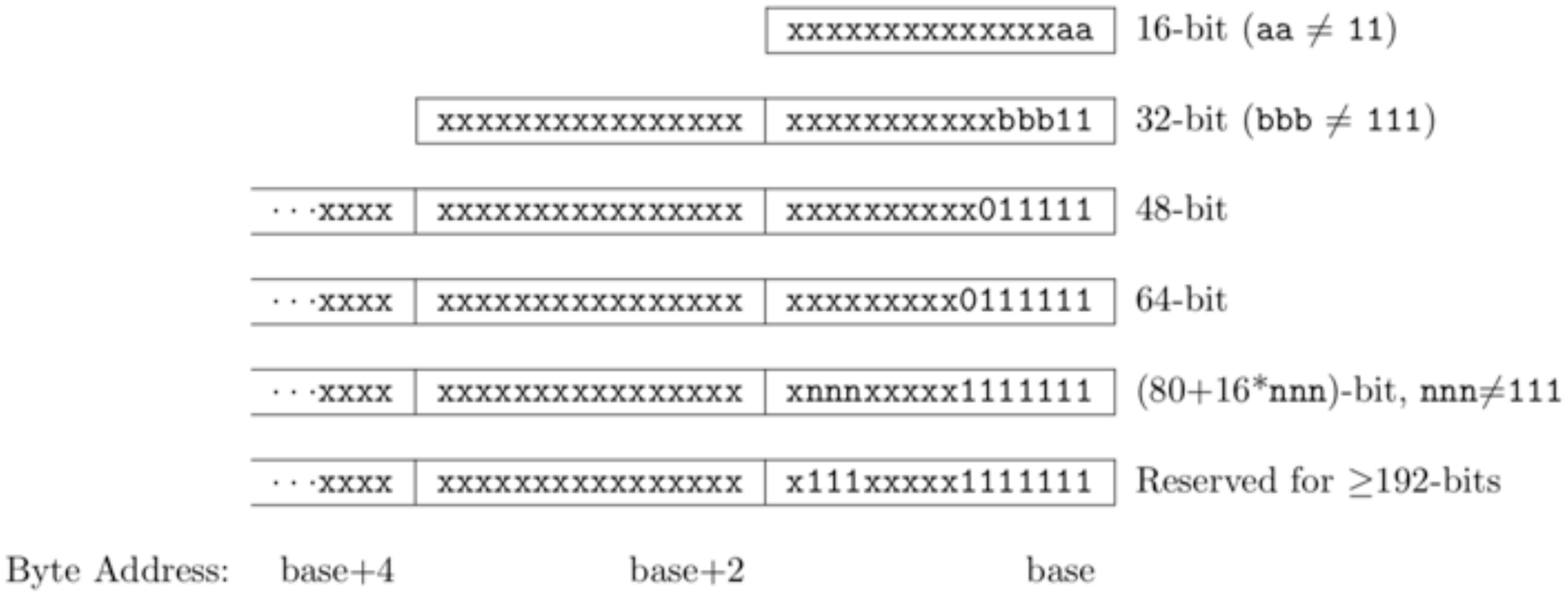
32x32-bit integer registers (**x0-x31**)
• **x0** always contains a 0

32 floating-point (FP) registers (**f0-f31**)
• each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)

FP status register (**fcsr**), used for FP rounding mode & exception reporting

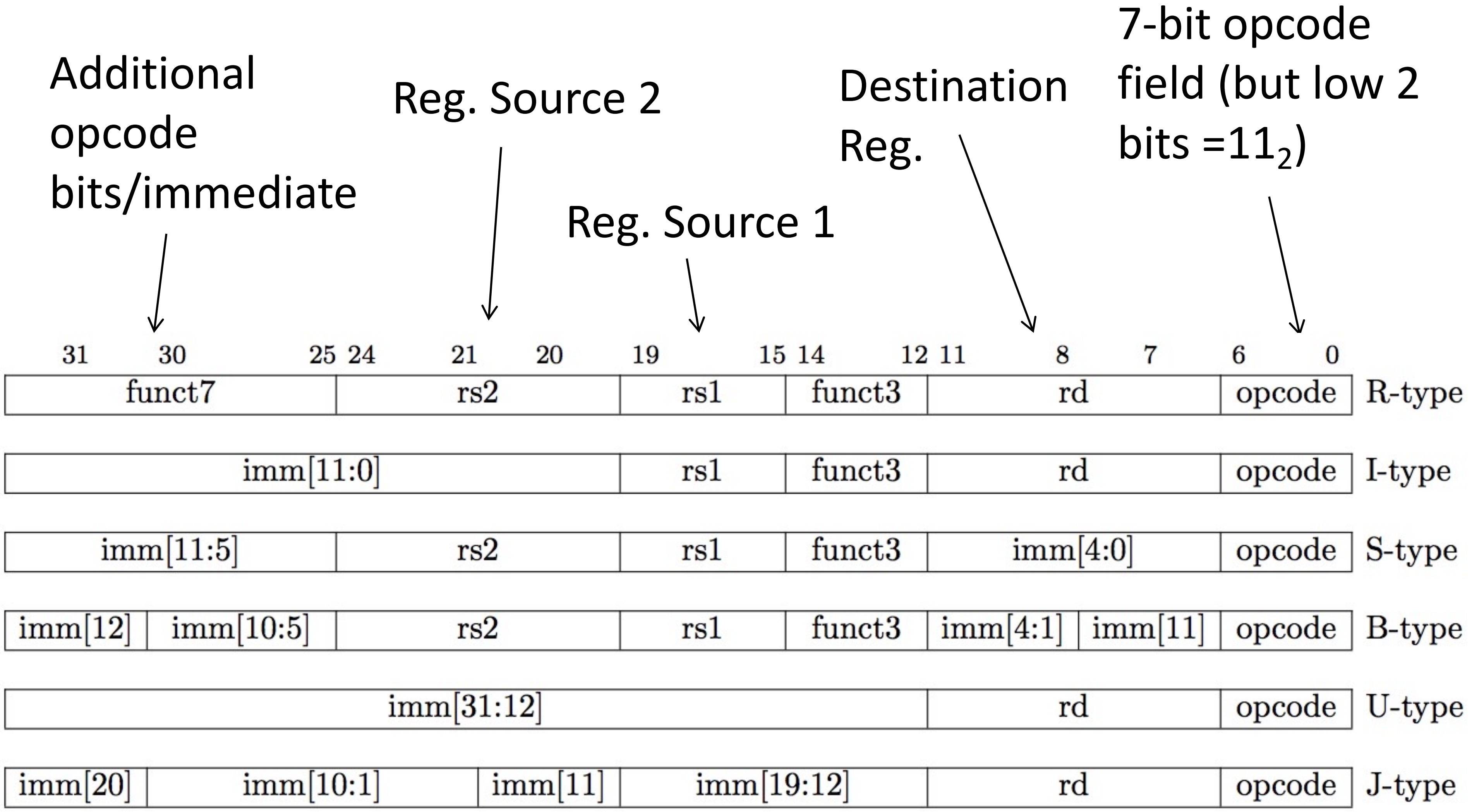


RISC-V Instruction Encoding

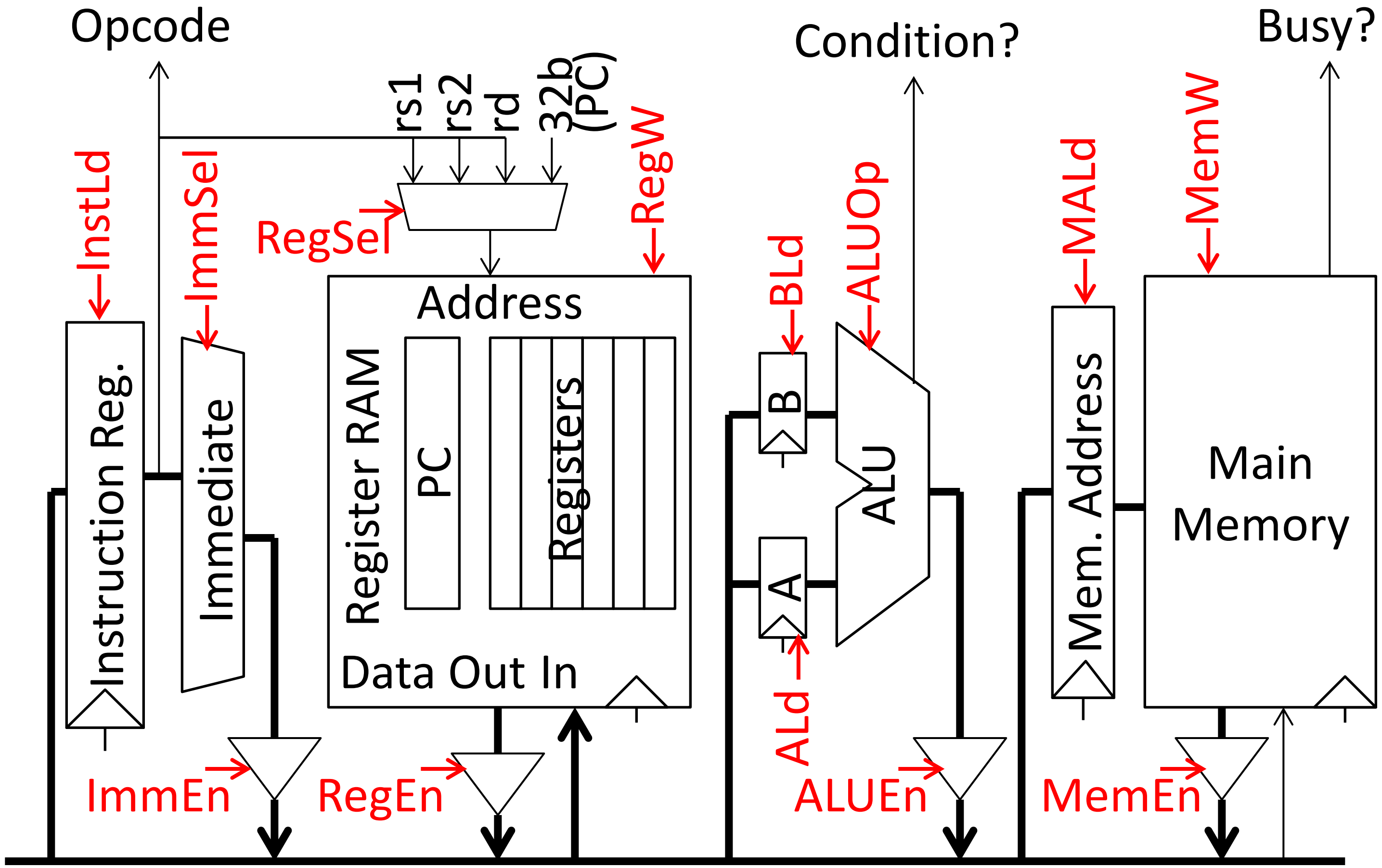


- Can support variable-length instructions.
- Base instruction set (RV32) always has fixed 32-bit instructions lowest two bits = 11_2
- All branches and jumps have targets at 16-bit granularity (even in base ISA where all instructions are fixed 32 bits)

RISC-V Instruction Formats



Single-Bus Datapath for Microcoded RISC-V



Microinstructions written as register transfers:

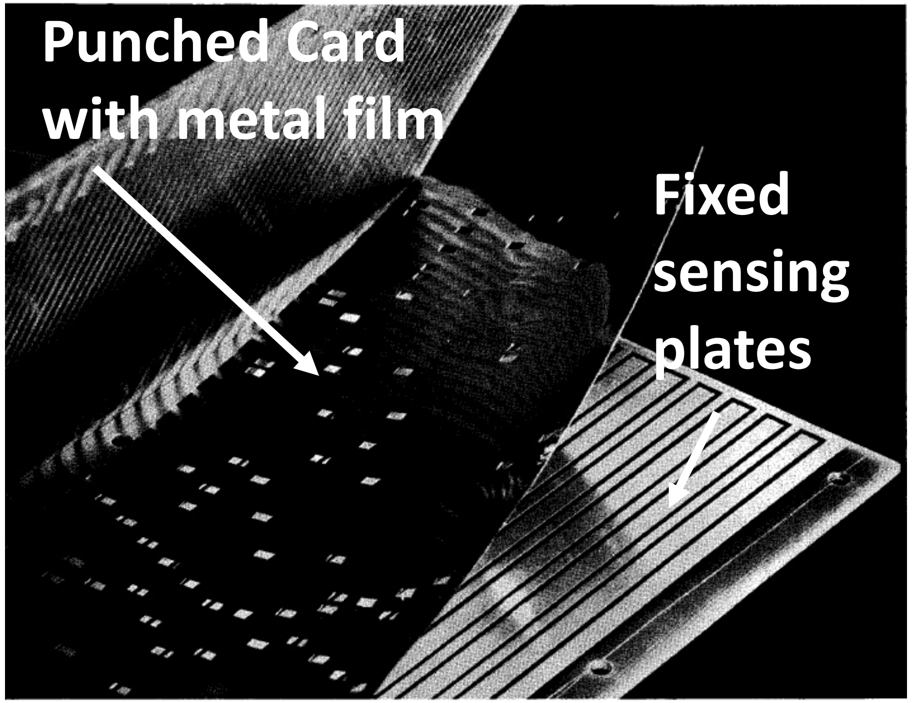
- MA:=PC means RegSel=PC; RegW=0; RegEn=1; MALd=1
- B:=Reg[rs2] means RegSel=rs2; RegW=0; RegEn=1; BLd=1
- Reg[rd]:=A+B means ALUop=Add; ALUEn=1; RegSel=rd; RegW=1

Inside Instruction Memory

Address				Data	
µPC	Opcode	Cond?	Busy?	Control Lines	Next µPC
fetch0	X	X	X	MA,A:=PC	fetch1
fetch1	X	X	1		fetch1
fetch1	X	X	0	IR:=Mem	fetch2
fetch2	ALU	X	X	PC:=A+4	ALU0
fetch2	ALUI	X	X	PC:=A+4	ALUI0
fetch2	LW	X	X	PC:=A+4	LW0
….					
ALU0	X	X	X	A:=Reg[rs1]	ALU1
ALU1	X	X	X	B:=Reg[rs2]	ALU2
ALU2	X	X	X	Reg[rd]:=ALUOp(A,B)	fetch0

Back Into History

IBM 360



IBM 360

ISA Compatible Computers

	M30	M40	M50	M65
Datapath width (bits)	8	16	32	64
µinst width (bits)	50	52	85	87
µcode size (K µinsts)	4	4	2.75	2.75
µstore cycle (ns)	750	625	500	200
memory cycle (ns)	1500	2500	2000	750
Rental fee ($K/month)	4	7	15	35

- Only the fastest models (75 and 95) were hardwired

Assembly Language
Snap Tutorial

Registers

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	t0	Temporary/alternate link register	Caller
x6–7	t1–2	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	[illegible]
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	[illegible]

Temporaries

Basic (Integer) Commends

Instruction Example	Description
lb t0, 8(sp)	Loads (dereferences) from memory address (sp + 8) into register t0. lb = load byte, lh = load halfword, lw = load word, ld = load doubleword.
sb t0, 8(sp)	Stores (dereferences) from register t0 into memory address (sp + 8). sb = store byte, sh = store halfword, sw = store word, sd = store doubleword.
add a0, t0, t1	Adds value of t0 to the value of t1 and stores the sum into a0.
addi a0, t0, -10	Adds value of t0 to the value -10 and stores the sum into a0.
sub a0, t0, t1	Subtracts value of t1 from value of t0 and stores the difference in a0.
mul a0, t0, t1	Multiplies the value of t0 to the value of t1 and stores the product in a0.
div a1, s3, t3	Dividies the value of t3 (denominator) from the value of s3 (numerator) and stores the quotient into the register a1.
rem a1, s3, t3	Divides the value of t3 (denominator) from the value of s3 (numerator) and stores the remainder into the register a1.

and a3, t3, s3	Performs logical AND on operands t3 and s3 and stores the result into the register a3.
or a3, t3, s3	Performs logical OR on operands t3 and s3 and stores the result into the register a3.
xor a3, t3, s3	Performs logical XOR on operands t3 and s3 and stores the result into the register a3.

sub a0, zero, a1

Translate: a0 = 0 – a1

Floating-Point Assembly

```
# Load a double-precision value
flw     ft0, 0(sp)
# ft0 now contains whatever we loaded from memory + 0
flw     ft1, 4(sp)
# ft1 now contains whatever we loaded from memory + 4
fadd.s  ft2, ft0, ft1
# ft2 is now ft0 + ft1
```

RISC-V supports floating-point

In fact, RISC-V has many modules:

Base	Version	Status
RVWMO	2.0	**Ratified**
RV32I	**2.1**	**Ratified**
RV64I	**2.1**	**Ratified**
RV32E	*1.9*	*Draft*
RV128I	*1.7*	*Draft*
Extension	Version	Status
M	**2.0**	**Ratified**
A	**2.1**	**Ratified**
F	**2.2**	**Ratified**
D	**2.2**	**Ratified**
Q	**2.2**	**Ratified**
C	**2.0**	**Ratified**
Counters	*2.0*	*Draft*
L	*0.0*	*Draft*
B	*0.0*	*Draft*
J	*0.0*	*Draft*
T	*0.0*	*Draft*
P	*0.2*	*Draft*
V	*0.7*	*Draft*
Zicsr	**2.0**	**Ratified**
Zifencei	**2.0**	**Ratified**
Zam	*0.1*	*Draft*
Ztso	*0.1*	*Frozen*

Branching

```c
for (int i = 0;i < 10;i++) {
    // Repeated code goes here.
}
```

```asm
# t0 = 0
li      t0, 0
li      t2, 10
loop_head:
bge     t0, t2, loop_end
# Repeated code goes here
addi    t0, t0, 1
j       loop_head
loop_end:
```

Example: Use the Stack

sp is a special register that is a stack

Statck: last in first out (LIFO)

```
addi    sp, sp, -8
sd      ra, 0(sp)
call    printf
ld      ra, 0(sp)
addi    sp, sp, 8
ret
```

C Function

```c
void my_function();
```

```
my_function:
        # Prologue
        addi    sp, sp, -32
        sd      ra, 0(sp)
        sd      a0, 8(sp)
        sd      s0, 16(sp)
        sd      s1, 24(sp)

        # Epilogue
        ld      ra, 0(sp)
        ld      a0, 8(sp)
        ld      s0, 16(sp)
        ld      s1, 24(sp)
        addi    sp, sp, 32
        ret
```

Good News is ⋯

- We have compiler that can convert C code into assembly
- 搭建RISC-V编译环境与运行环境 - 知乎 (zhihu.com)
- riscv-collab/riscv-gnu-toolchain: GNU toolchain for RISC-V, including GCC (github.com)

Hardware Code Examples

- Counter
- Finite State Machine
- Memory
  - https://bonany.gitlab.io/pis/
- Arithmetic Logic Units