Computer Architecture

Week 6: RISC-V Processor

Fenerbahçe University

Professor & TAs

Prof: Dr. Vecdi Emre Levent

Office: 311

Email: emre.levent@fbu.edu.tr

TA: Arş. Gör. Uğur Özbalkan

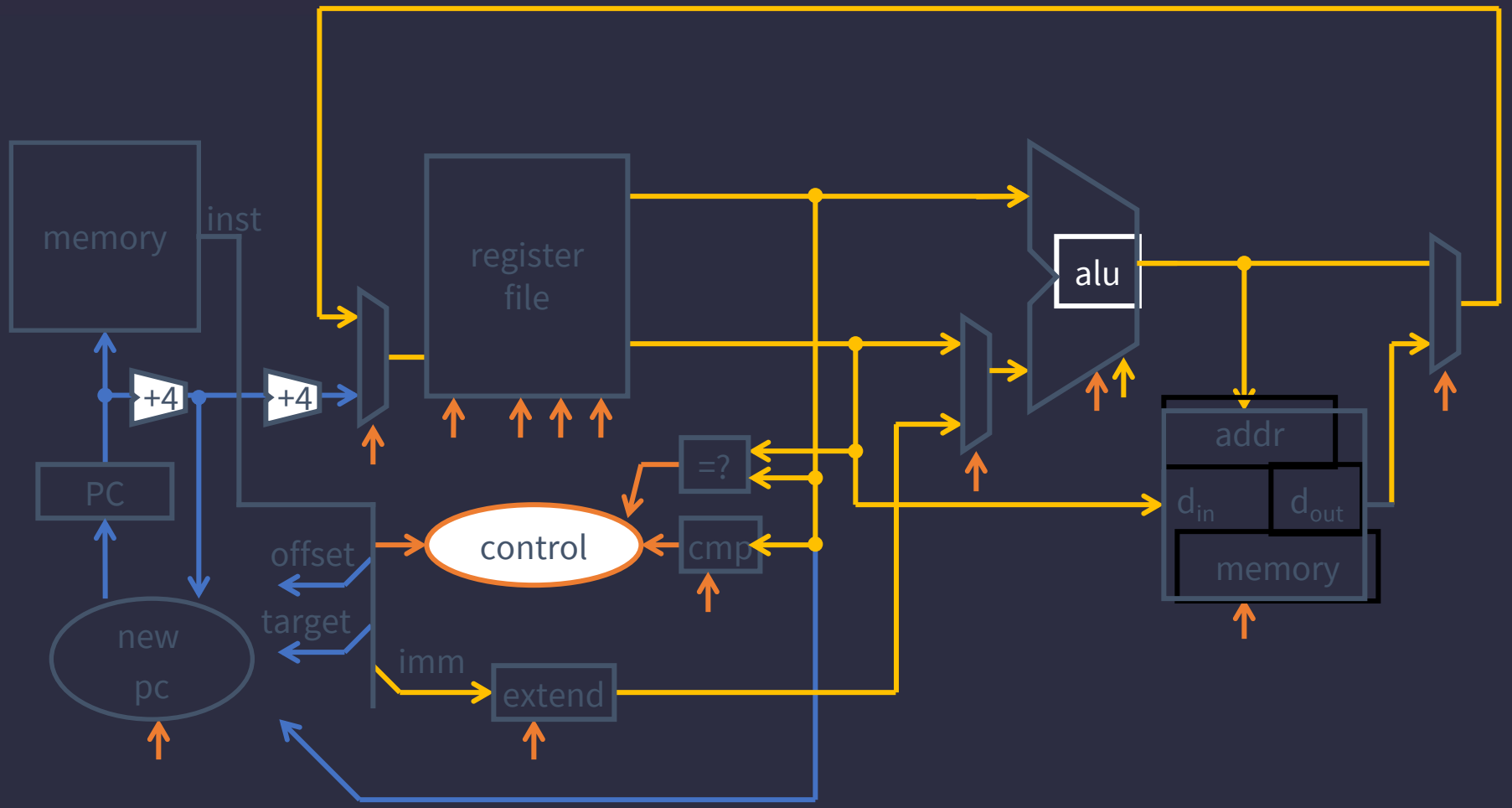
Office: 311

Email: ugur.ozbalkan@fbu.edu.tr

Course Plan

- RISC-V Processor

Processor Building



A single cycle processor

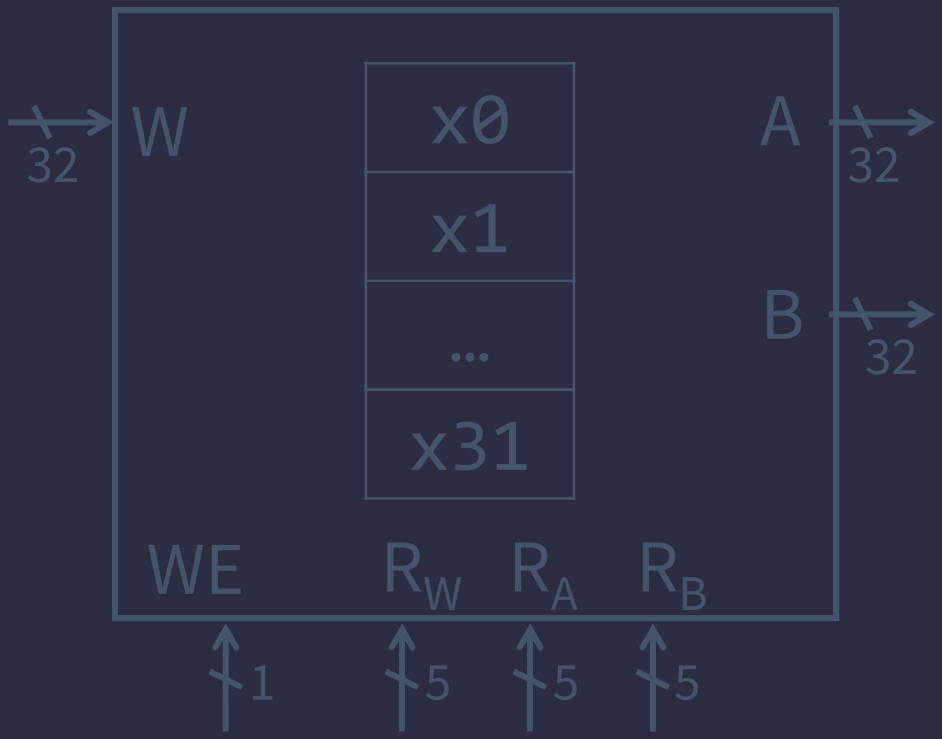
Goal for the lectures

- Understanding the basics of a processor
  - We now have the technology to build a CPU!

- Putting it all together:
  - Arithmetic Logic Unit (ALU)
  - Register File
  - Memory
    - SRAM: cache
    - DRAM: main memory
  - RISC-V Instructions & how they are executed

RISC-V Register File

- RISC-V register file
  - 32 registers, 32-bits each
  - x0 wired to zero
  - Write port indexed via R_W
    - on falling edge when WE=1
  - Read ports indexed via R_A, R_B



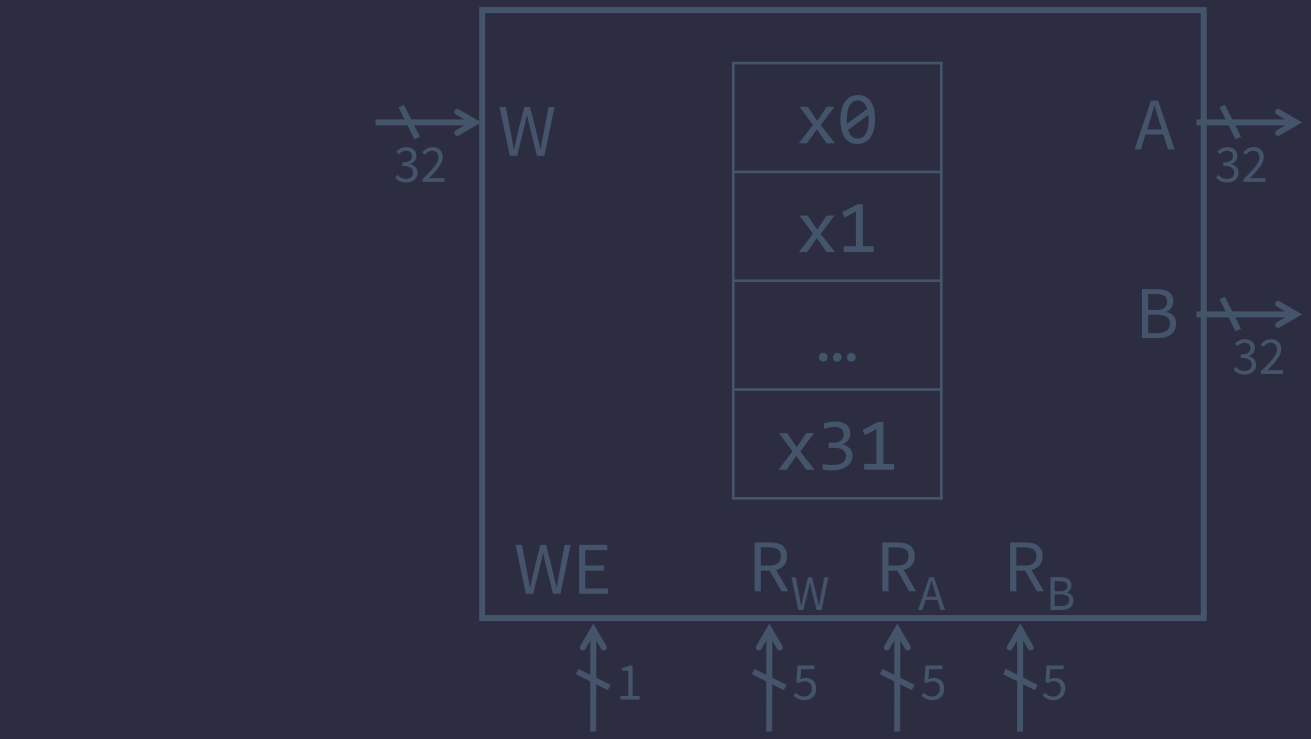
- RISC-V register file
  - Numbered from 0 to 31
  - Can be referred by number: x0, x1, x2, … x31
  - Convention, each register also has a name:
    - x10 – x17 → a0 – a7, x28 – x31 → t3 – t6

64 bit support

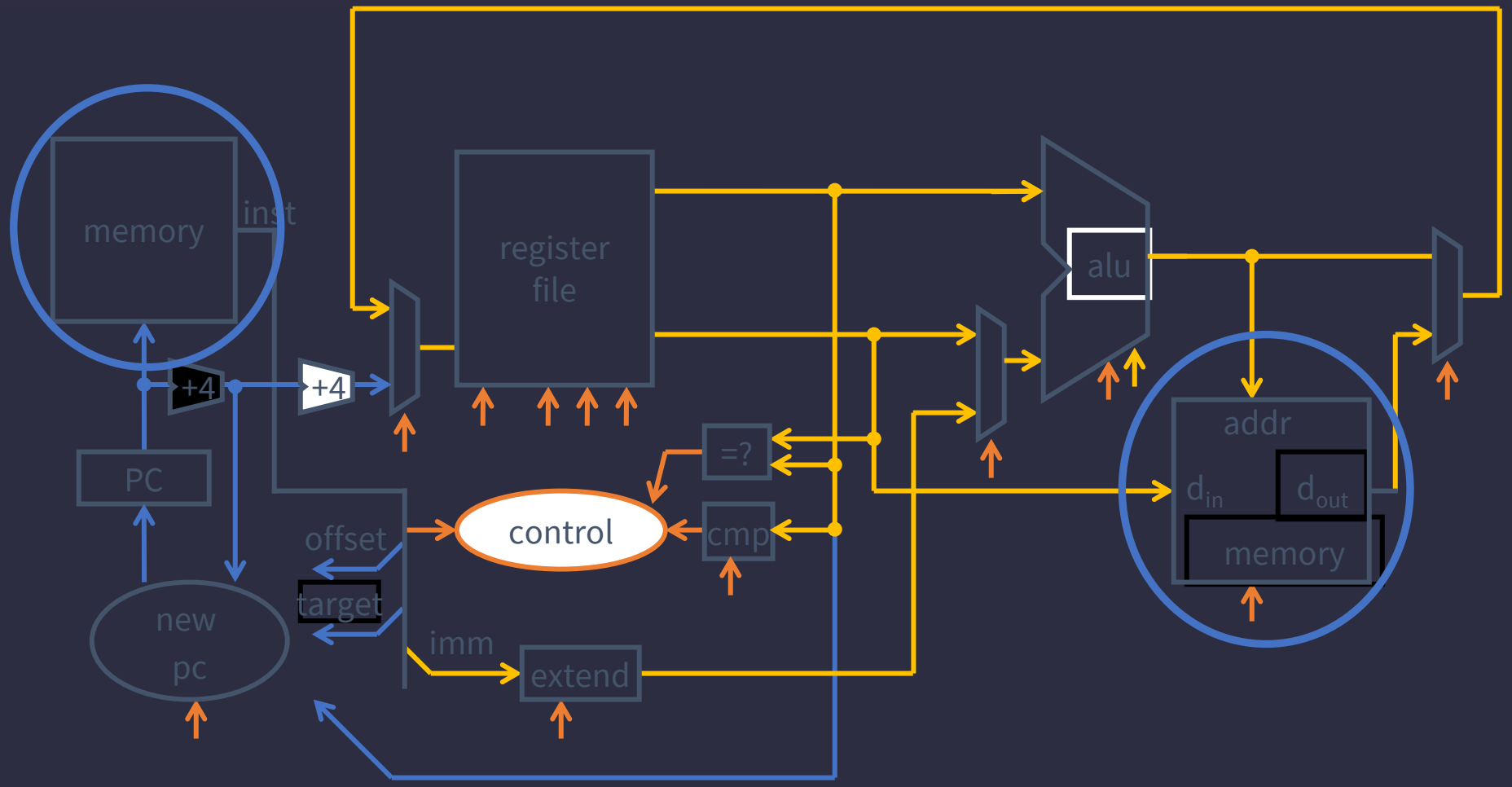
If we wanted to support 64 registers, what would change?

R_W, R_A, R_B

5 → 6



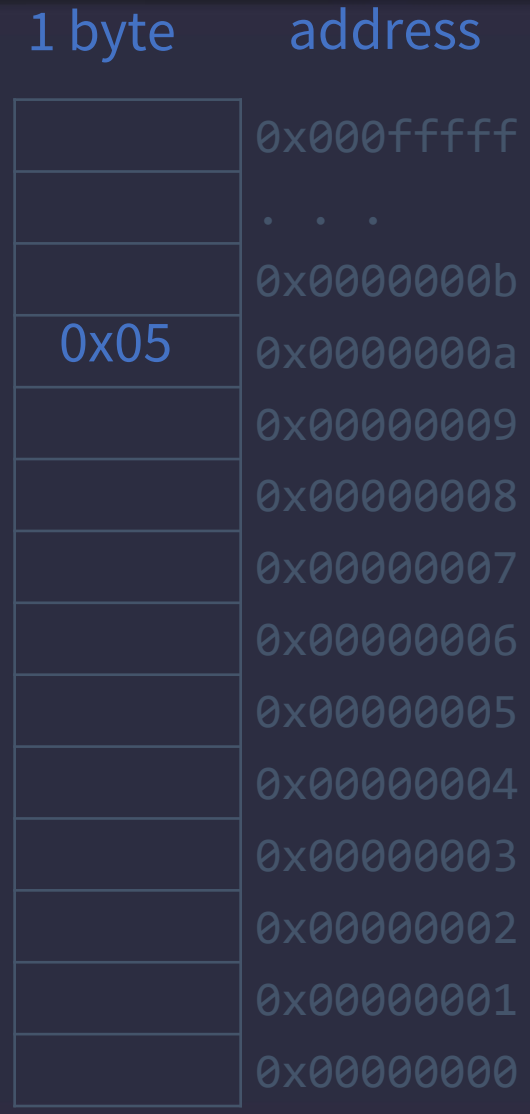
RISC-V Memory



A single cycle processor

RISC-V Memory

- 32-bit address
- 32-bit data
- Enable + 2 bit memory control (mc)



00: read word (4 byte aligned)

01: write byte

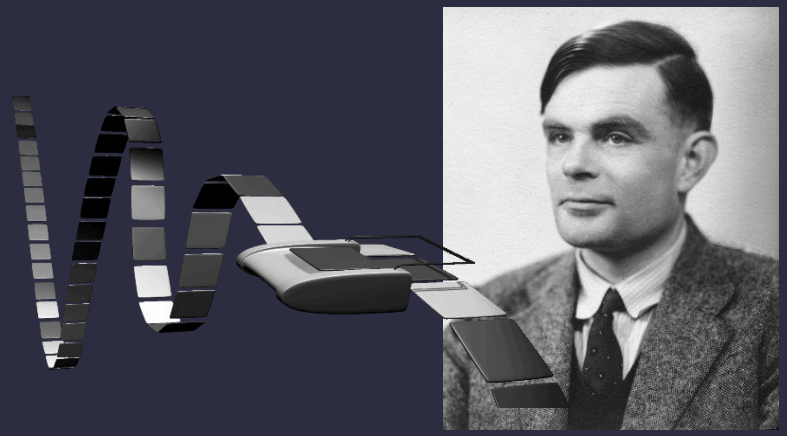
10: write halfword (2 byte aligned)

11: write word (4 byte aligned)

To make a computer

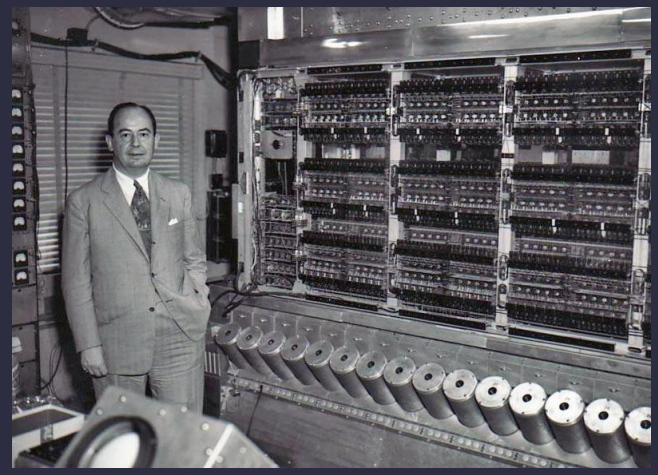
Need a program

- Stored program computer
- (a Universal Turing Machine)



Architectures

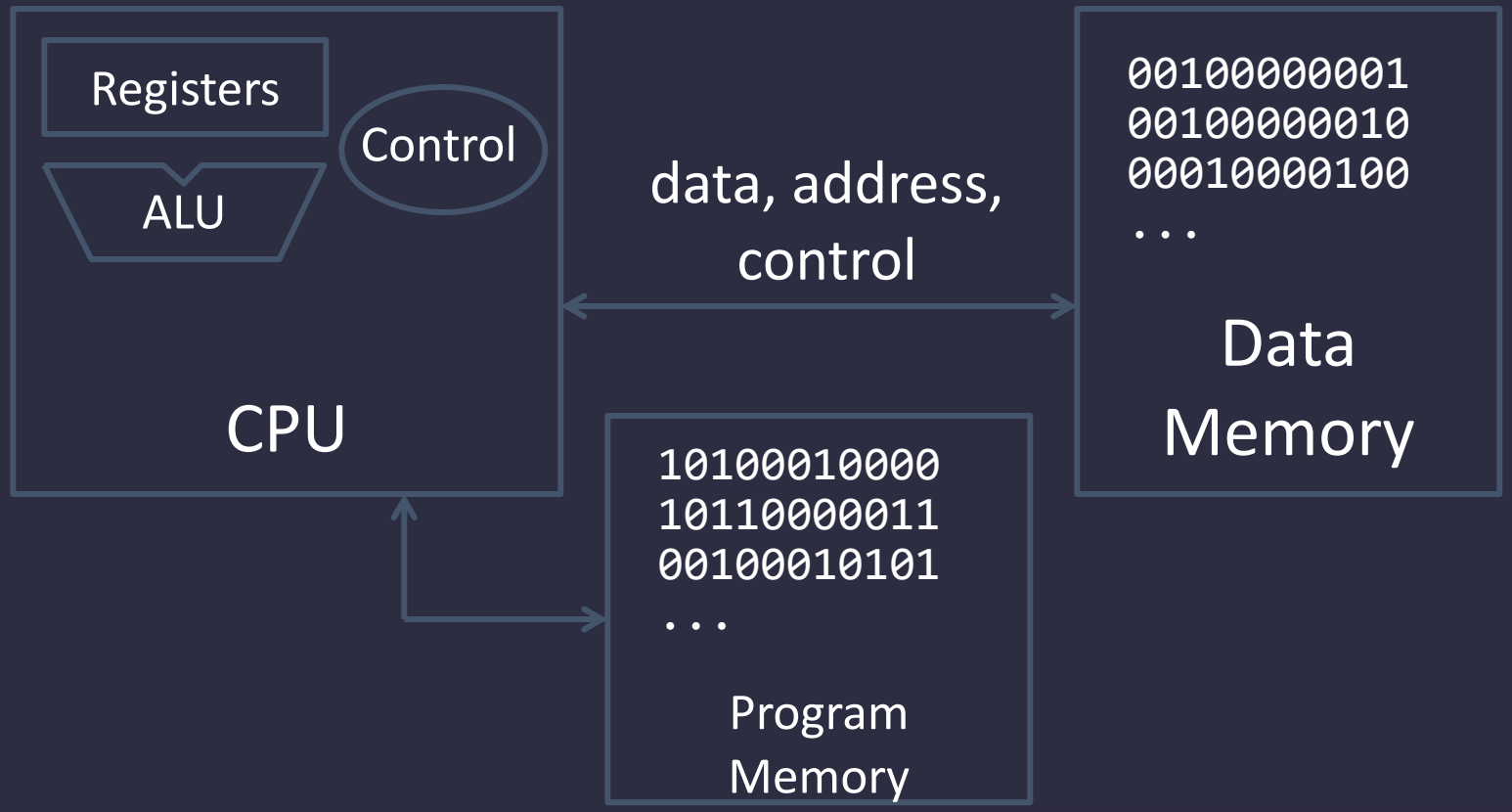
- von Neumann architecture
- Harvard (modified) architecture



Putting it all together: Basic Processor

A RISC-V CPU with a (modified) Harvard architecture

- Modified: instructions & data in common address space, separate instr/data caches can be accessed in parallel



Next Goal

- How to program and execute instructions on a RISC-V processor?

Instruction Usage

Instructions are stored in memory, encoded in binary

A basic processor

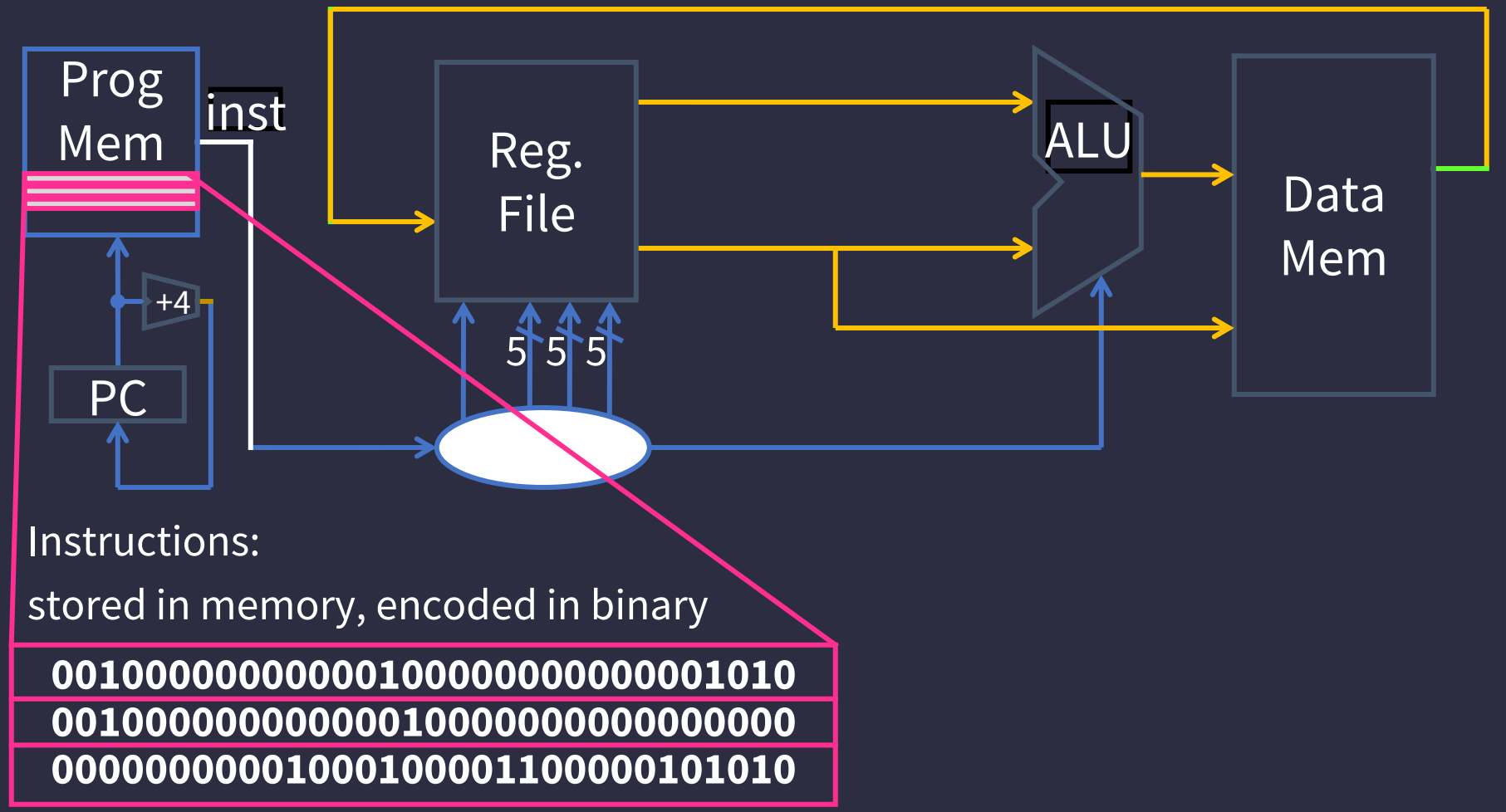
- fetches
- decodes
- executes

one instruction at a time

Instruction Processing

A basic processor
- fetches
- decodes
- executes

one instruction at a time



Levels of Interpretation: Instructions

```
for (i = 0; i < 10; i++)
        printf("test");
```

```
main:   addi x2, x0, 10
        addi x1, x0, 0
loop:   slt x3, x1, x2
        ...
```

10 x2 x0 op=addi

```
00000000101000010000000000010011
00100000000000010000000000010000
00000000001000100001100000101010
```

Instruction Set Architecture

ALU, Control, Register File, …

Assembly Language

- No symbols (except labels)
- One operation per statement
- "human readable machine language"

Machine Language

- Binary-encoded assembly
- Labels become addresses
- **The language of the CPU**

Machine Implementation (Microarchitecture)

Instruction Set Architecture (ISA)

Different CPU architectures specify different instructions

Two classes of ISAs

- Reduced Instruction Set Computers (RISC)
  IBM Power PC, Sun Sparc, MIPS, Alpha
- Complex Instruction Set Computers (CISC)
  Intel x86, PDP-11, VAX

Another ISA classification: Load/Store Architecture

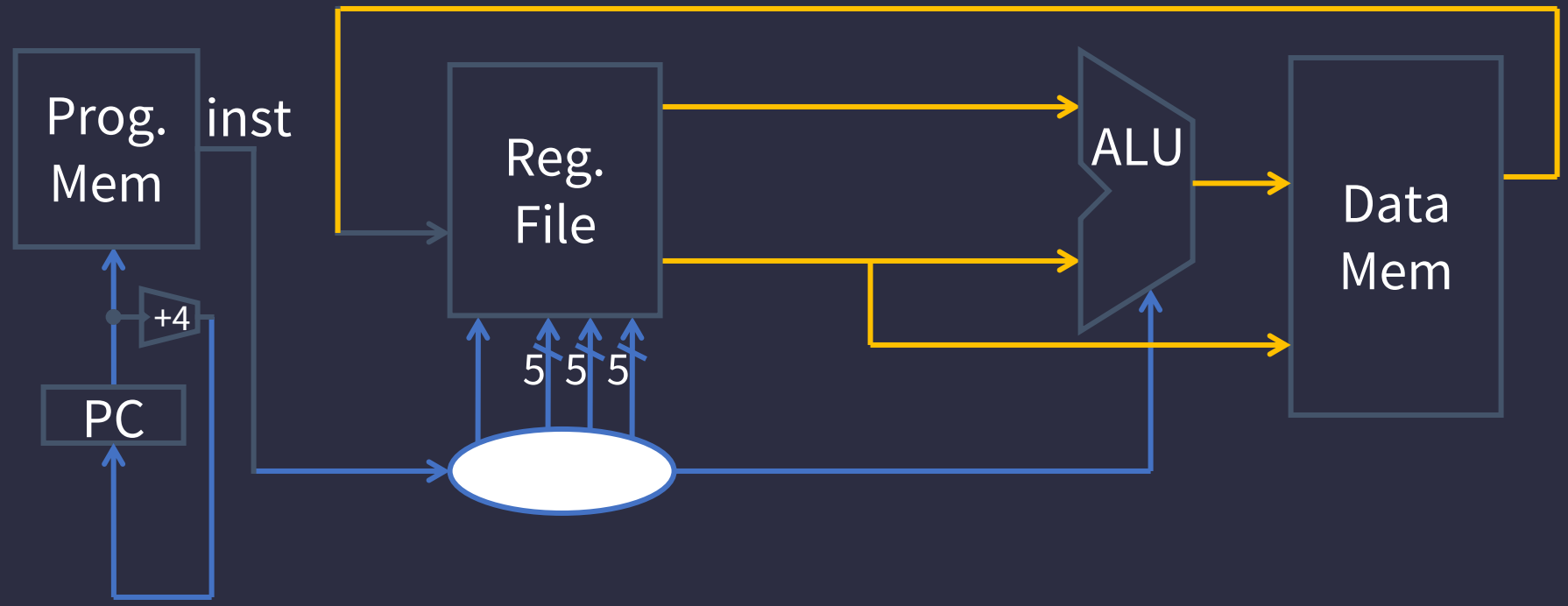
- Data must be in registers to be operated on
  For example: array[x] = array[y] + array[z]
  1 add ? OR 2 loads, an add, and a store ?
- Keeps HW simple → many RISC ISAs are load/store

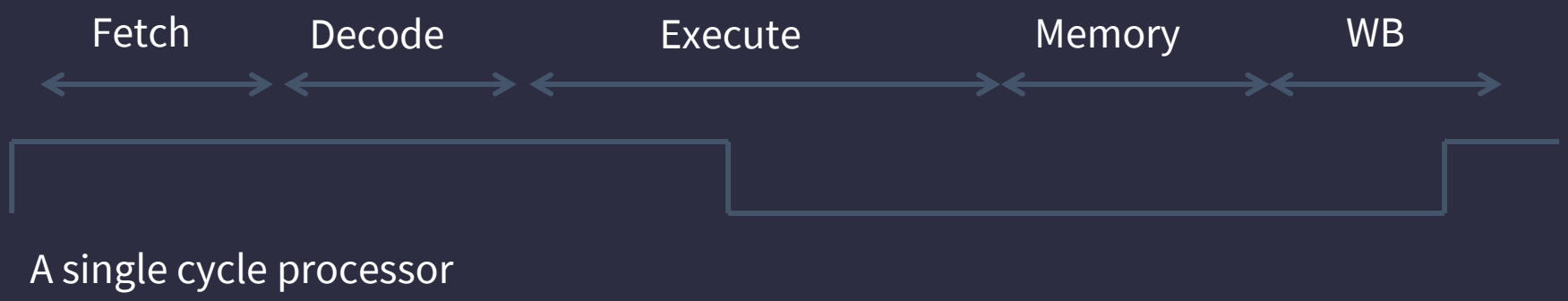
Next Goal

How are instructions executed?

What is the general datapath to execute an instruction?

Five Stages of RISC-V Datapath





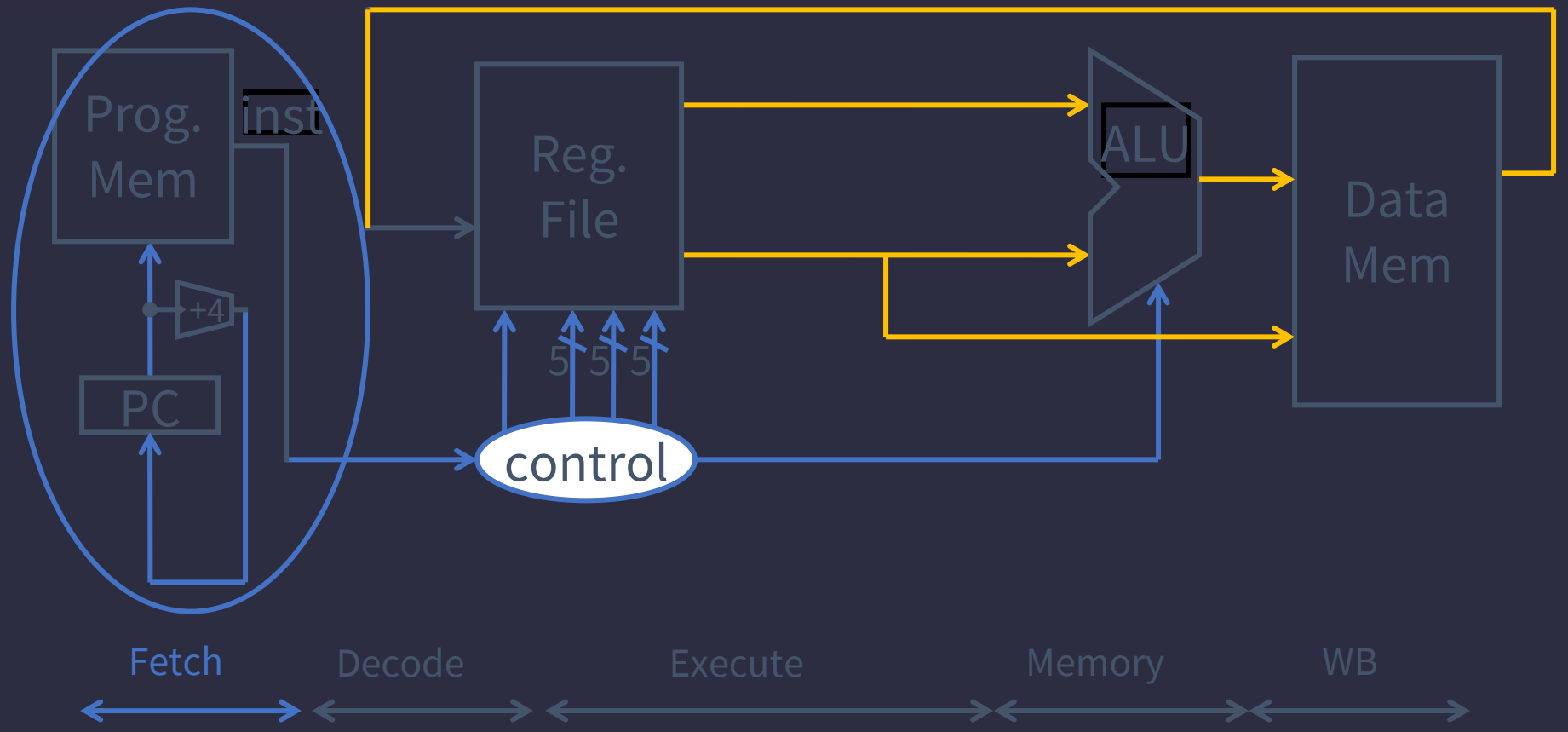
A single cycle processor

Five Stages of RISC-V Datapath

Basic CPU execution loop

1. Instruction Fetch
2. Instruction Decode
3. Execution (ALU)
4. Memory Access
5. Register Writeback

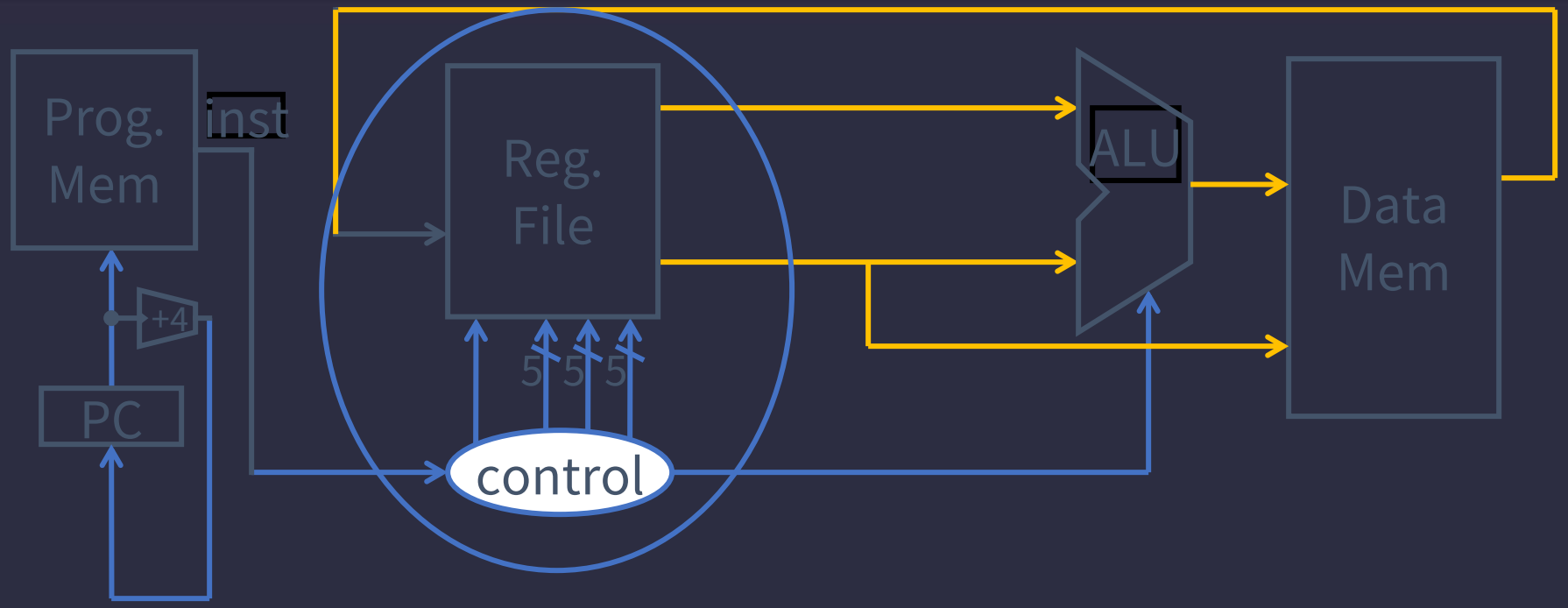
Stage 1: Instruction Fetch



Fetch 32-bit instruction from memory

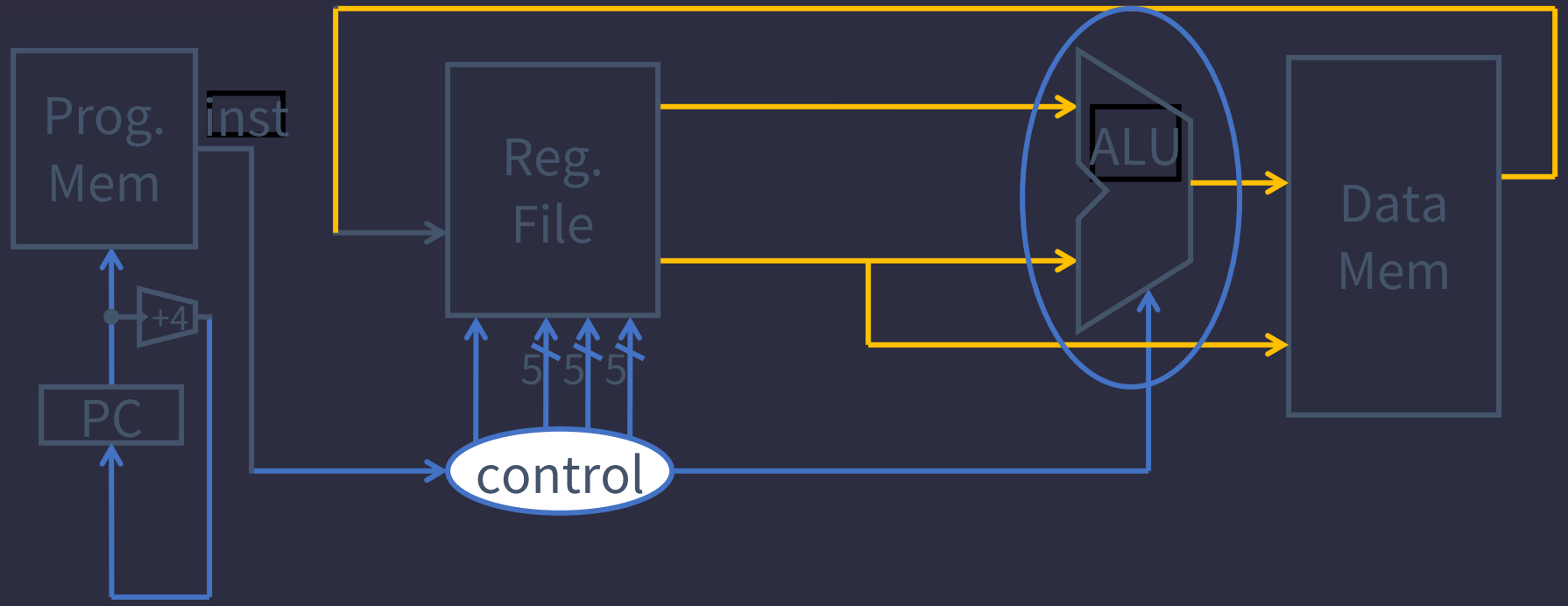
Increment PC = PC + 4

Stage 2: Instruction Decode



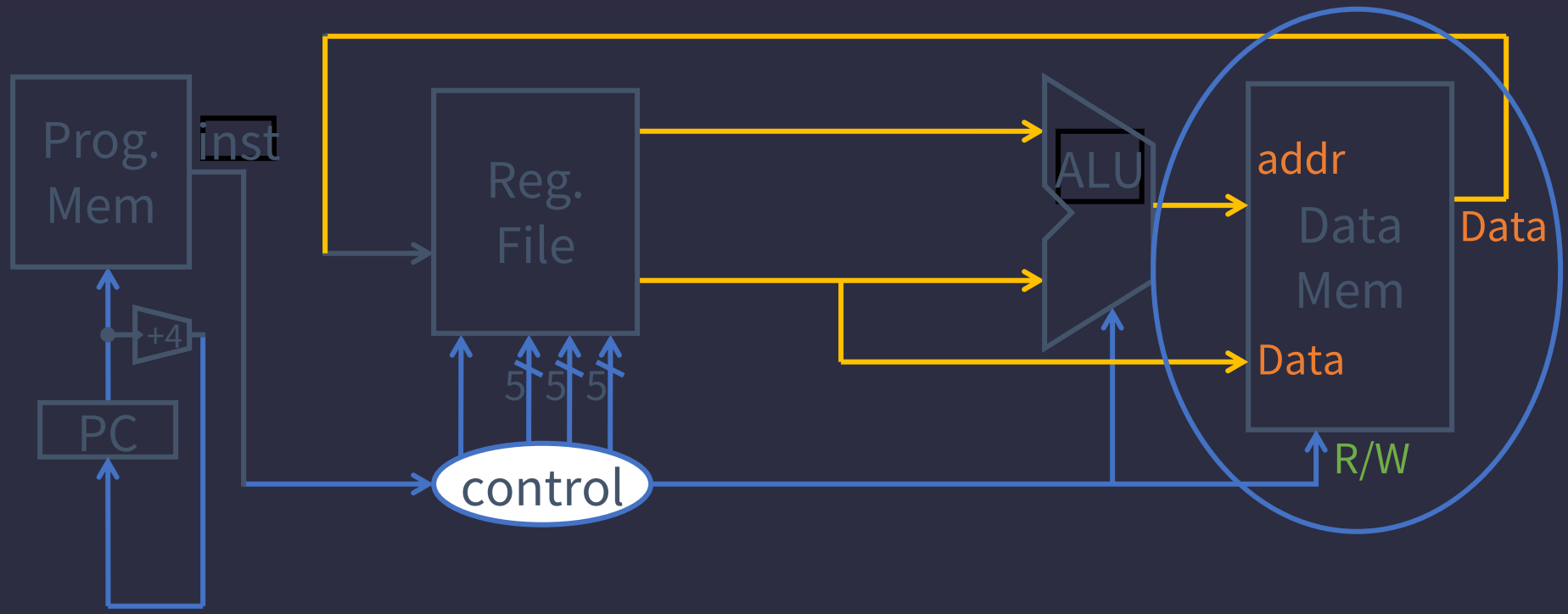
Gather data from the instruction
Read opcode; determine instruction type, field lengths
Read in data from register file
(0, 1, or 2 reads for `jump`, `addi`, or `add`, respectively)

Stage 3: Execution (ALU)



Useful work done here (+, -, *, /), shift, logic operation, comparison (slt)
Load/Store? lw x2, x3, 32 → Compute address

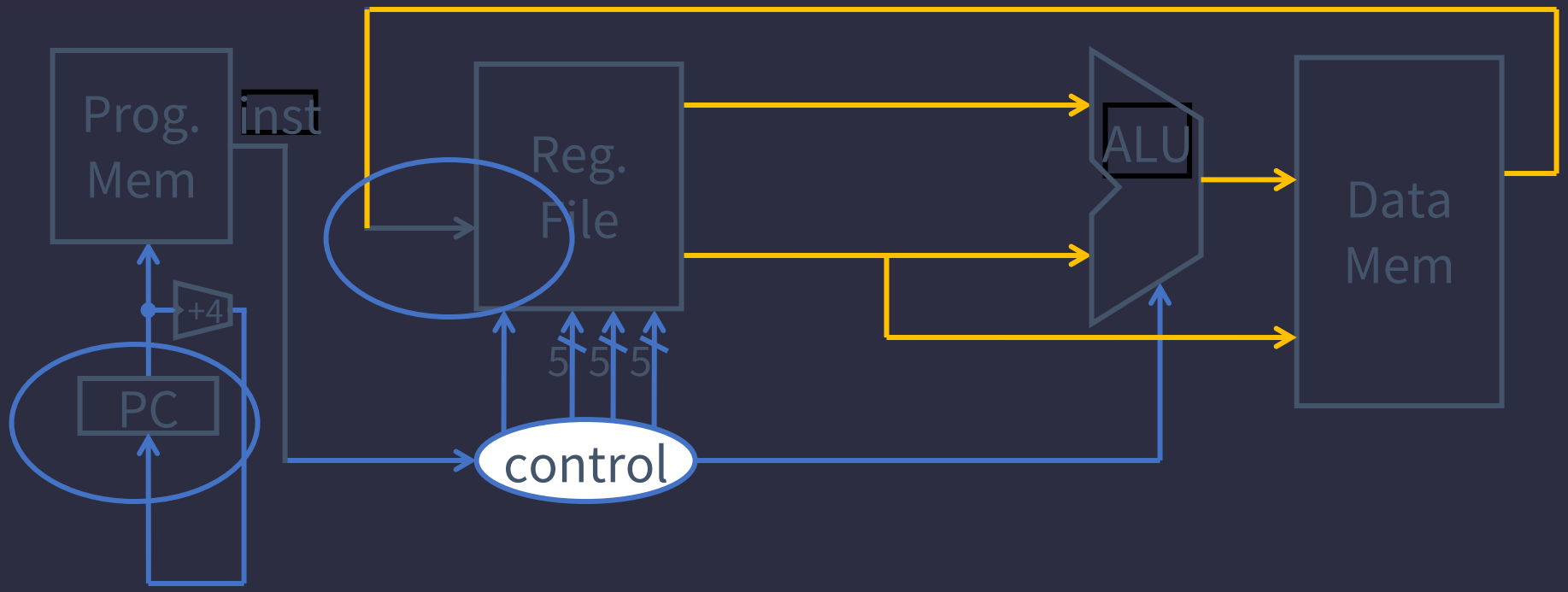
Stage 4: Memory Access



Used by load and store data only

Instructions will skip this stage

Stage 5: Writeback



Write to register file

- For arithmetic ops, logic, shift, etc, load. What about stores?

Update PC

- For branches, jumps

Summary

- The datapath for a RISC-V processor has five stages:
  1. Instruction Fetch
  2. Instruction Decode
  3. Execution (ALU)
  4. Memory Access
  5. Register Writeback

- This five stage datapath is used to execute all RISC-V instructions

Next Goal

- Specific datapaths RISC-V Instructions

RISC-V Design Principles

Simplicity favors regularity

- 32 bit instructions

Smaller is faster

- Small register file

Instruction Types

- Arithmetic
  - add, subtract, shift left, shift right, multiply, divide
- Memory
  - load value from memory to a register
  - store value to memory from a register
- Control flow
  - conditional jumps (branches)
  - jump and link (subroutine call)

RISC-V Instruction Types

- Arithmetic/Logical
  - R-type: result and two source registers, shift amount
  - I-type: result and source register, shift amount in 16-bit immediate with sign/zero extension
  - U-type: result register, 16-bit immediate with sign/zero extension
- Memory Access
  - I-type for loads and S-type for stores
  - load/store between registers and memory
  - word, half-word and byte operations
- Control flow
  - UJ-type: jump-and-link
  - I-type: jump-and-link register
  - SB-type: conditional branches: pc-relative addresses

RISC-V instruction formats

All RISC-V instructions are 32 bits long

32-bit RISC-V Instruction Formats

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper Immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

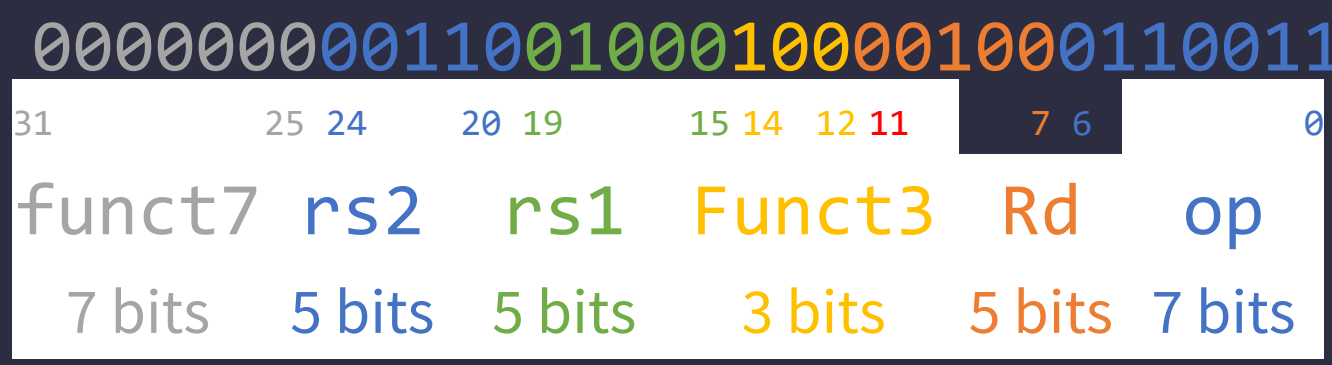
- **opcode (7 bit):** partially specifies which of the 6 types of *instruction formats*
- **funct7 + funct3 (10 bit):** combined with **opcode**, these two fields describe what operation to perform
- **rs1 (5 bit):** specifies register containing first operand
- **rs2 (5 bit):** specifies second register operand
- **rd (5 bit):**: Destination register specifies register which will receive result of computation

RISC-V instruction formats

All RISC-V instructions are 32 bits long

Tip	Komut	Kod[3]	Kod[7]	İşlem
Opcode: 0x33 **Register-Register** (R-Type)	add rd, rs1, rs2	0x0	0x00	R[rd] ← R[rs1] + R[rs2]
	mul rd, rs1, rs2	0x0	0x01	R[rd] ← (R[rs1] * R[rs2])[31:0]
	sub rd, rs1, rs2	0x0	0x20	R[rd] ← R[rs1] - R[rs2]
	sll rd, rs1, rs2	0x1	0x00	R[rd] ← R[rs1] << R[rs2]
	mulh rd, rs1, rs2	0x1	0x01	R[rd] ← (R[rs1] * R[rs2])[63:32]
	slt rd, rs1, rs2	0x2	0x00	R[rd] ← (R[rs1] < R[rs2]) ? 1 : 0 (signed)
	xor rd, rs1, rs2	0x4	0x00	R[rd] ← R[rs1] ^ R[rs2]
	div rd, rs1, rs2	0x4	0x01	R[rd] ← R[rs1] / R[rs2]
	srl rd, rs1, rs2	0x5	0x00	R[rd] ← R[rs1] >> R[rs2]
	or rd, rs1 rs2	0x6	0x00	R[rd] ← R[rs1] | R[rs2]
	rem rd, rs1, rs2	0x6	0x01	R[rd] ← R[rs1] % R[rs2]
	and rd, rs1, rs2	0x7	0x00	R[rd] ← R[rs1] & R[rs2]
Opcode: 0x03 **Immediate (Değer)** (I-Type)	lb rd, offset(rs1)	0x0		R[rd] ← Mem(R[rs1] + offset) (byte) (SignedExtend)
	lh rd, offset(rs1)	0x1		R[rd] ← Mem(R[rs1] + offset) (half) (SignedExtend)
	lw rd, offset(rs1)	0x2		R[rd] ← Mem(R[rs1] + offset) (word)
Opcode: 0x13 **Immediate (Değer)** (I-Type)	addi rd, rs1, value	0x0	0x00	R[rd] ← R[rs1] + value
	slli rd, rs1, value	0x1	0x00	R[rd] ← R[rs1] << value
	slti rd, rs1, value	0x2	0x00	R[rd] ← (R[rs1] < value) ? 1 : 0 (signed)
	xori rd, rs1, value	0x4	0x00	R[rd] ← R[rs1] ^ value
	srli rd, rs1, value	0x5	0x00	R[rd] ← R[rs1] >> value
	ori rd, rs1, value	0x6	0x00	R[rd] ← R[rs1] | value
	andi rd, rs1, value	0x7	0x00	R[rd] ← R[rs1] & value
Opcode: 0x23 **Store (Kayıt)** (S-Type)	sw rs2, offset(rs1)	0x2		Mem(R[rs1] + offset) ← R[rs2]
	swge rs2, 0(rs1), offset	0x7		if(R[rs2] >= offset(signed)) Mem(R[rs1] + offset) ← R[rs2]
Opcode: 0x63 **Branch (Dallanma)** (B-Type)	beq rs1, rs2, offset	0x0		if(R[rs1] == R[rs2]) PC ← PC + offset
	blt rs1, rs2, offset	0x4		if(R[rs1] < R[rs2]) (signed) PC ← PC + offset
	bltu rs1, rs2, offset	0x6		if(R[rs1] < R[rs2]) (unsigned) PC ← PC + offset
Opcode: 0x37 **Upper Immediate** (U-Type)	lui rd, offset			R[rd] ← offset
Opcode: 0x67 **Upper Immediate(Atlama)** (UJ-Type)	jal rd, value			R[rd] ← PC + 4 PC ← PC + value
Opcode: 0x6F **Immediate(Atlama)** (IJ-Type)	jalr rd, rs, value			R[rd] ← PC + 4 PC ← R[rs] + value

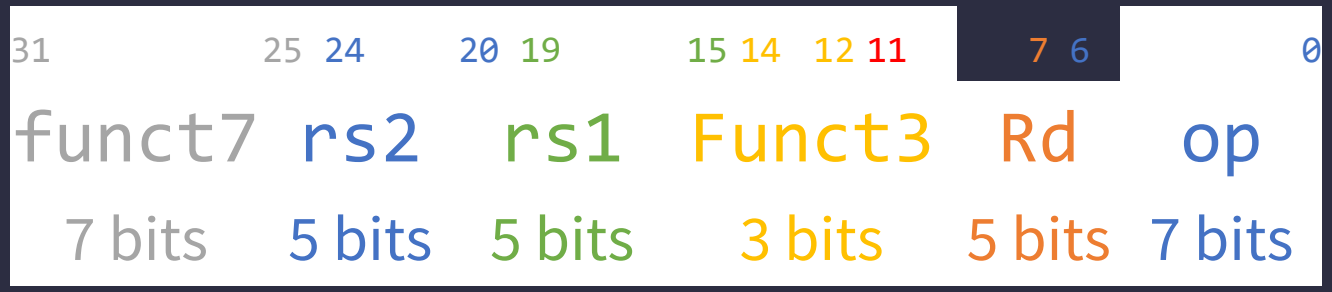
R-Type (1): Arithmetic and Logic



op	funct3	mnemonic	description
0110011	000	ADD rd, rs1, rs2	R[rd] = R[rs1] + R[rs2]
0110011	000	SUB rd, rs1, rs2	R[rd] = R[rs1] – R[rs2]
0110011	110	OR rd, rs1, rs2	R[rd] = R[rs1] | R[rs2]
0110011	100	XOR rd, rs1, rs2	R[rd] = R[rs1] ⊕ R[rs2]

R-Type (1): Arithmetic and Logic

```
00000000011001000100001000110011
```



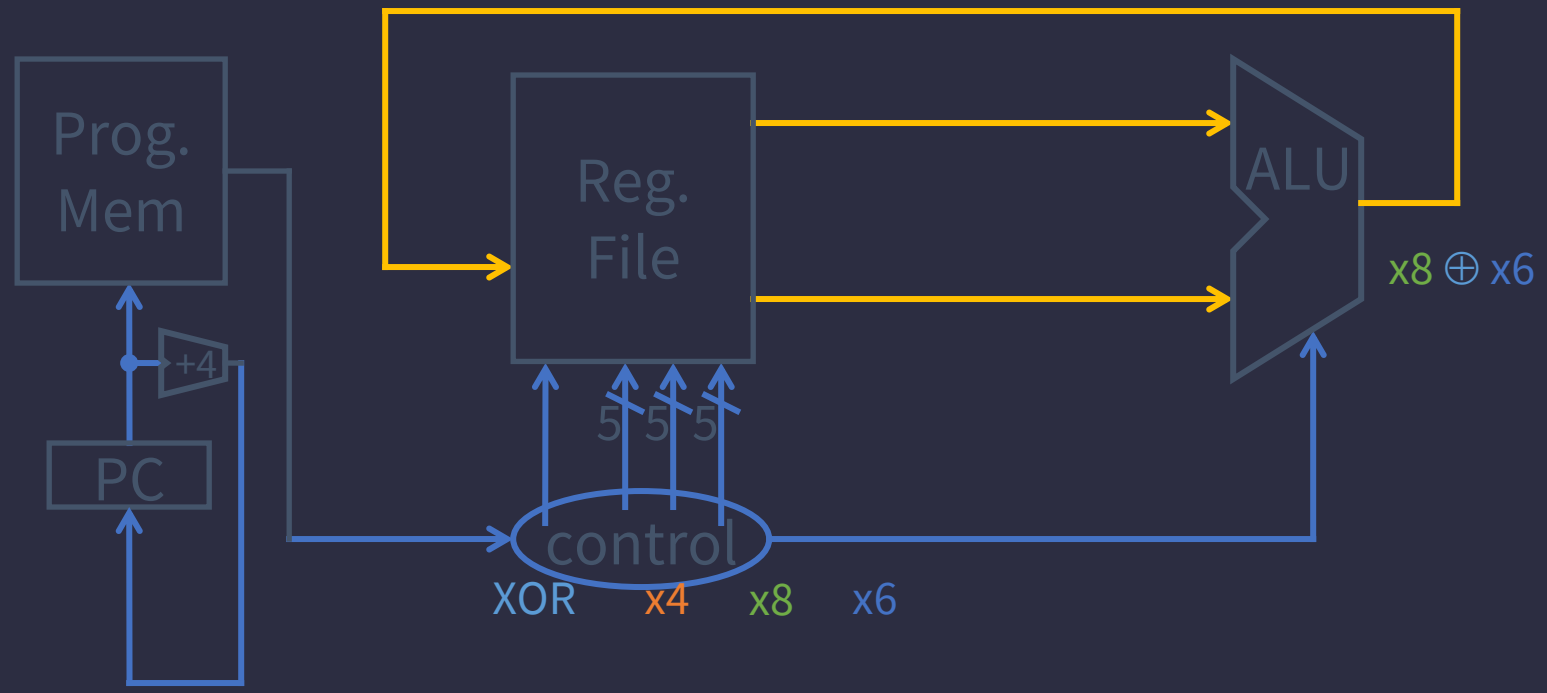
op	funct3	mnemonic	description
0110011	000	ADD rd, rs1, rs2	R[rd] = R[rs1] + R[rs2]
0110011	000	SUB rd, rs1, rs2	R[rd] = R[rs1] – R[rs2]
0110011	110	OR rd, rs1, rs2	R[rd] = R[rs1] | R[rs2]
0110011	100	XOR rd, rs1, rs2	R[rd] = R[rs1] ⊕ R[rs2]

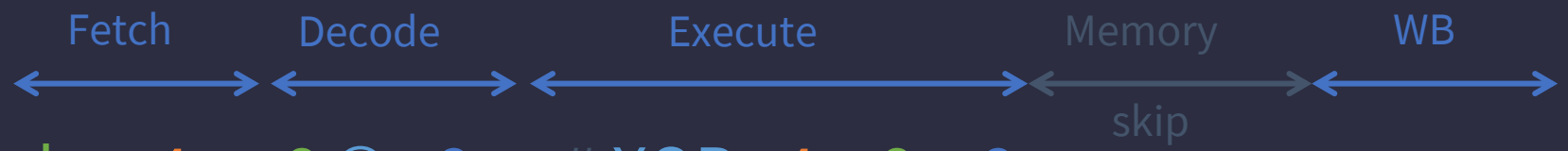
Example: x4 = x8 ⊕ x6 # XOR x4, x8, x6

rd, rs1, rs2

Arithmetic and Logic

XOR x4, x8, x6

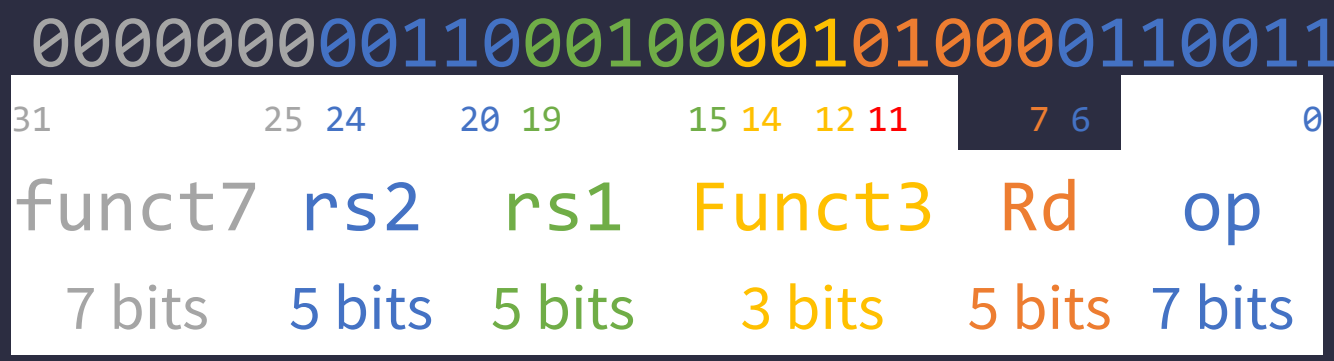




Example: x4 = x8 ⊕ x6 # XOR x4, x8, x6

rd, rs1, rs2

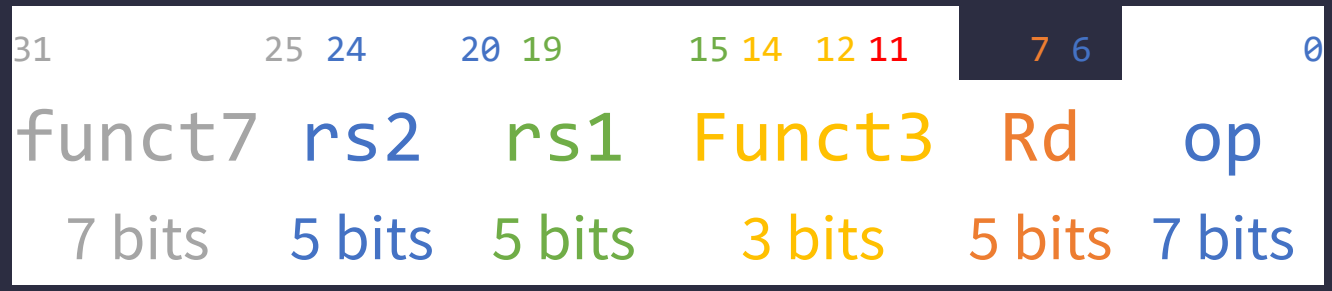
R-Type (2): Shift Instructions



op	funct3	mnemonic	description
0110011	001	SLL rd, rs1, rs2	R[rd] = R[rs1] << R[rs2]
0110011	101	SRL rd, rs1, rs2	R[rd] = R[rs1] >>> R[rs2] (zero ext.)
0110011	101	SRA rd, rs1, rs2	R[rd] = R[rt] >>> R[rs2] (sign ext.)

R-Type (2): Shift Instructions

```
0000000 00110 00100 001 01000 0110011
```

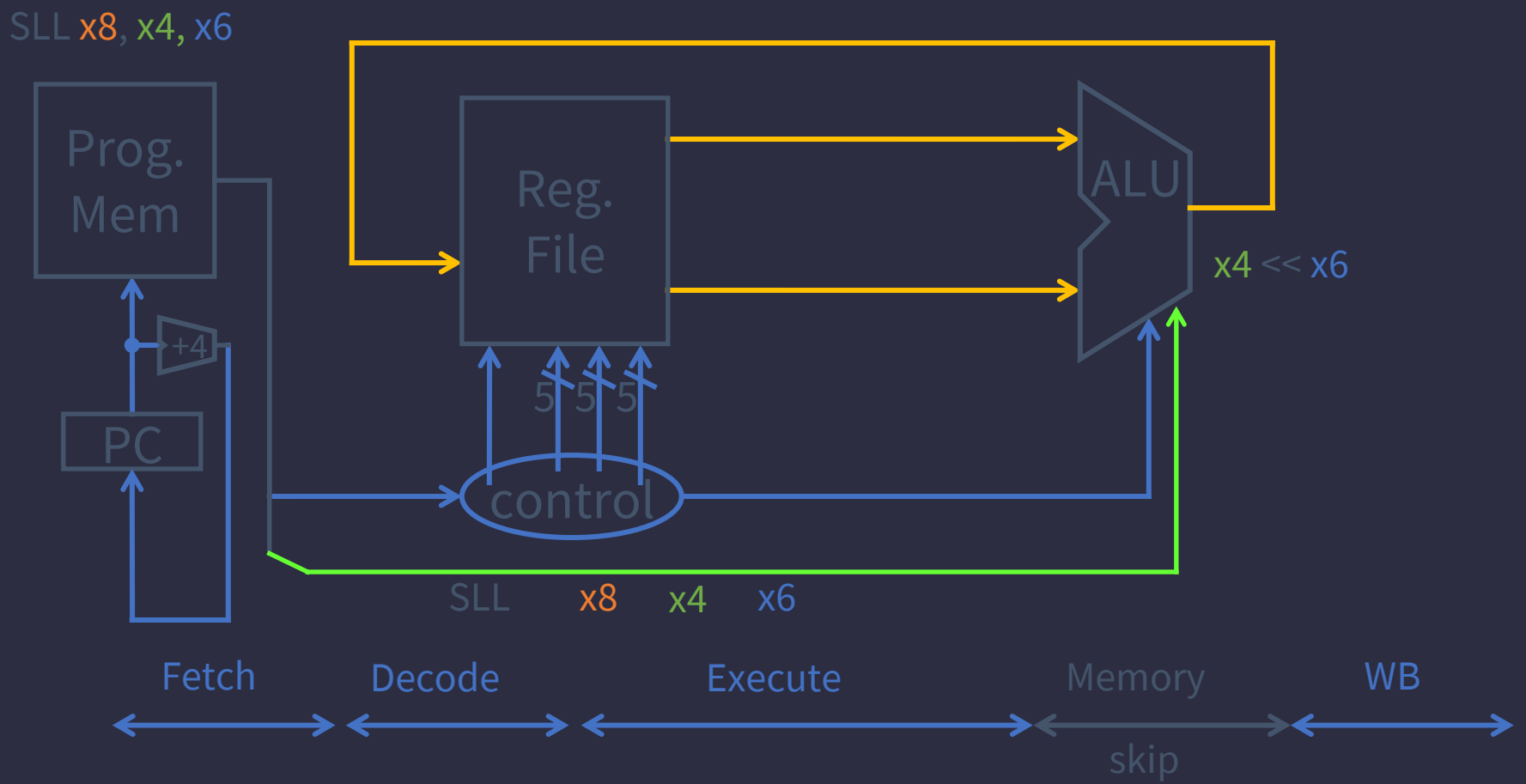


op	funct3	mnemonic	description
0110011	001	SLL rd, rs1, rs2	R[rd] = R[rs1] << R[rs2]
0110011	101	SRL rd, rs1, rs2	R[rd] = R[rs1] >>> R[rs2] (zero ext.)
0110011	101	SRA rd, rs1, rs2	R[rd] = R[rt] >>> R[rs2] (sign ext.)

Example: x8 = x4 * 2^{x6} # SLL x8, x4, x6

x8 = x4 << x6

Shift



Example: x8 = x4 * 2^{x6} # SLL x8, x4, x6

x8 = x4 << x6

I-Type (1): Arithmetic w/ immediates

op	funct3	mnemonic	description
0010011	000	ADDI rd, rs1, imm	R[rd] = R[rs1] + sign_extend(imm)
0010011	111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
0010011	110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

I-Type (1): Arithmetic w/ immediates

```
00000000010100101000001010010011
31                  20 19    15 14  12 11     7 6          0
```

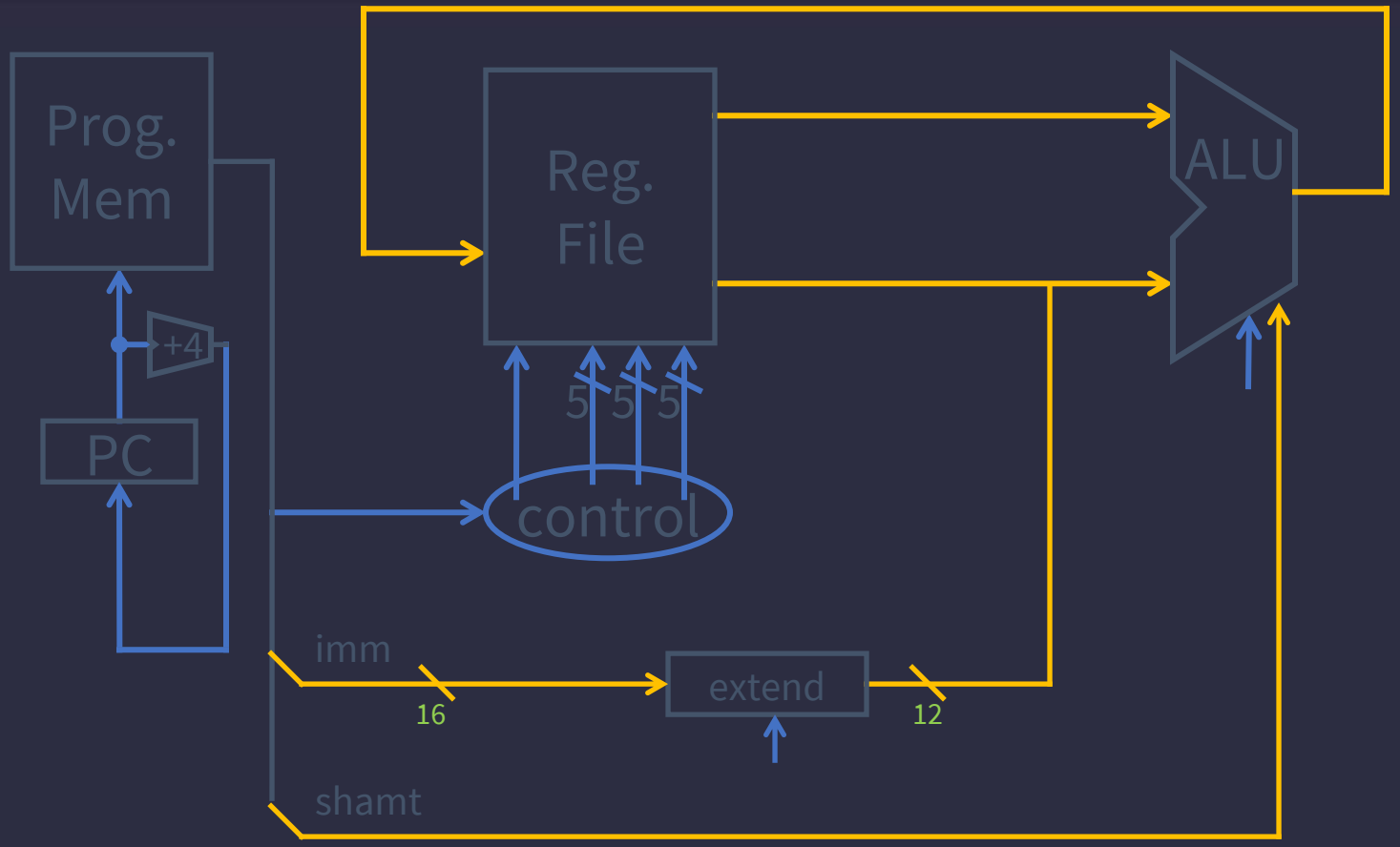
imm	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

op	funct3	mnemonic	description
0010011	000	ADDI rd, rs1, imm	R[rd] = R[rs1] + sign_extend(imm)
0010011	111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
0010011	110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

Example: x5 = x5 + 5 # ADDI x5, x5, 5

x5 += 5

Arithmetic w/ immediates

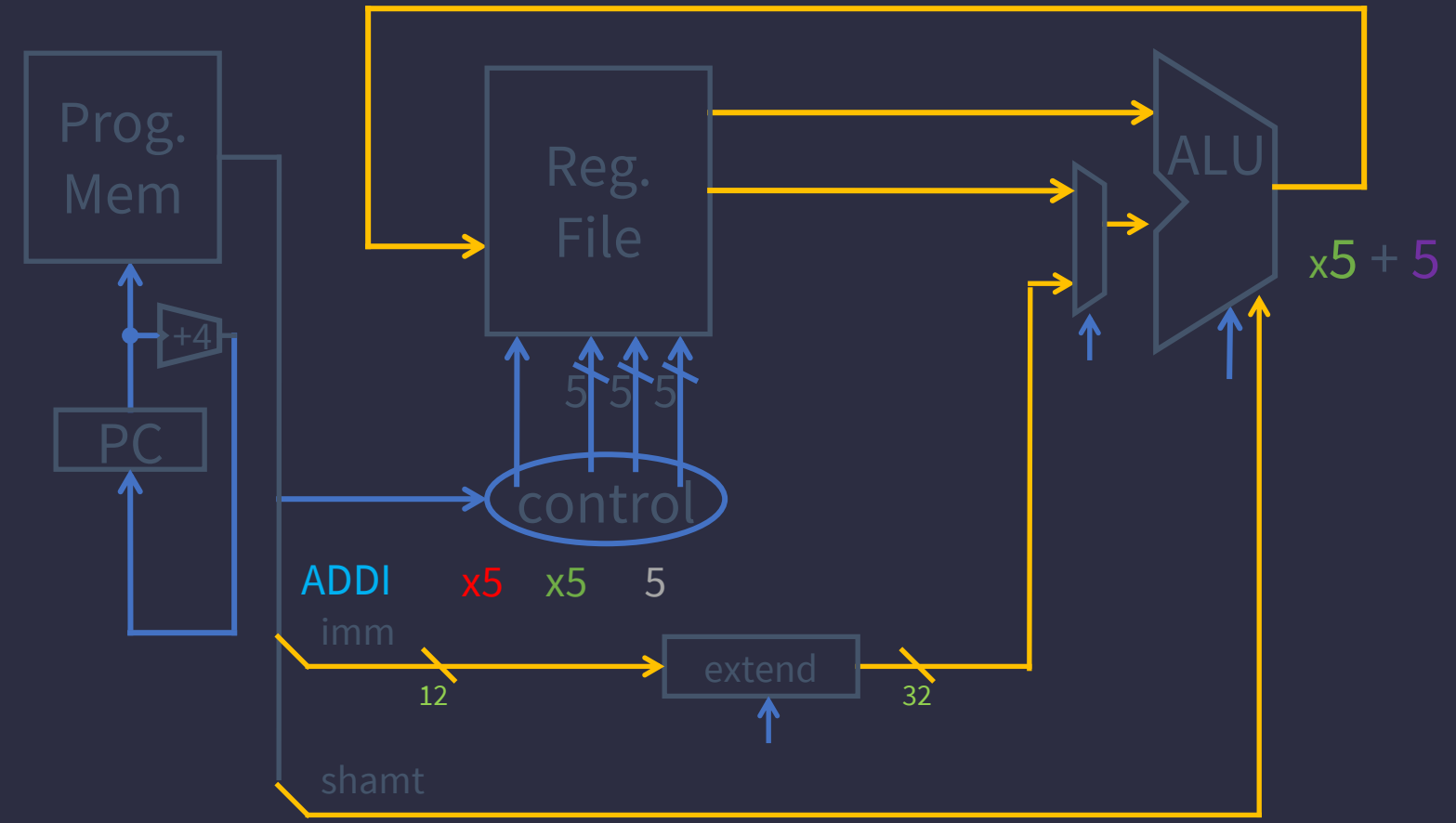


Example: x5 = x5 + 5 # ADDI x5, x5, 5

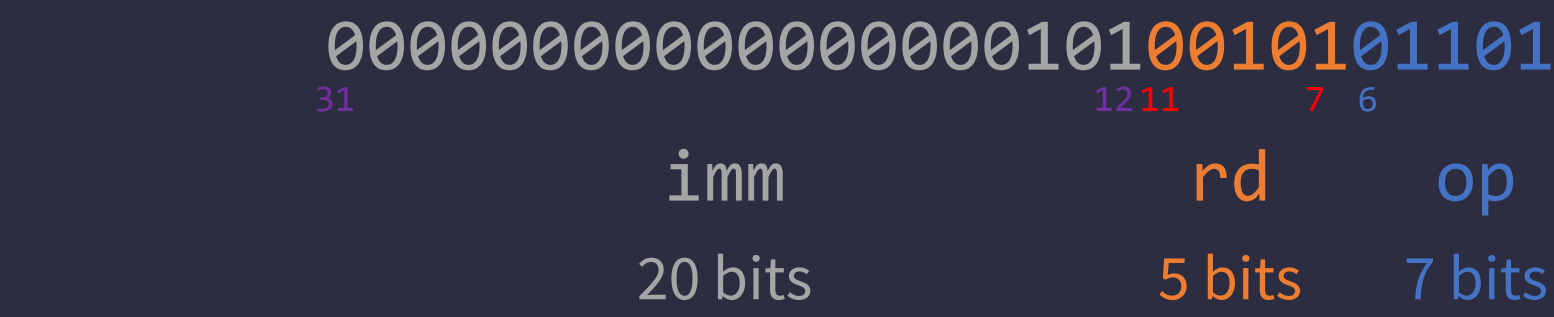
Arithmetic w/ immediates

ADDI x5, x5, 5

Example: x5 = x5 + 5 # ADDI x5, x5, 5

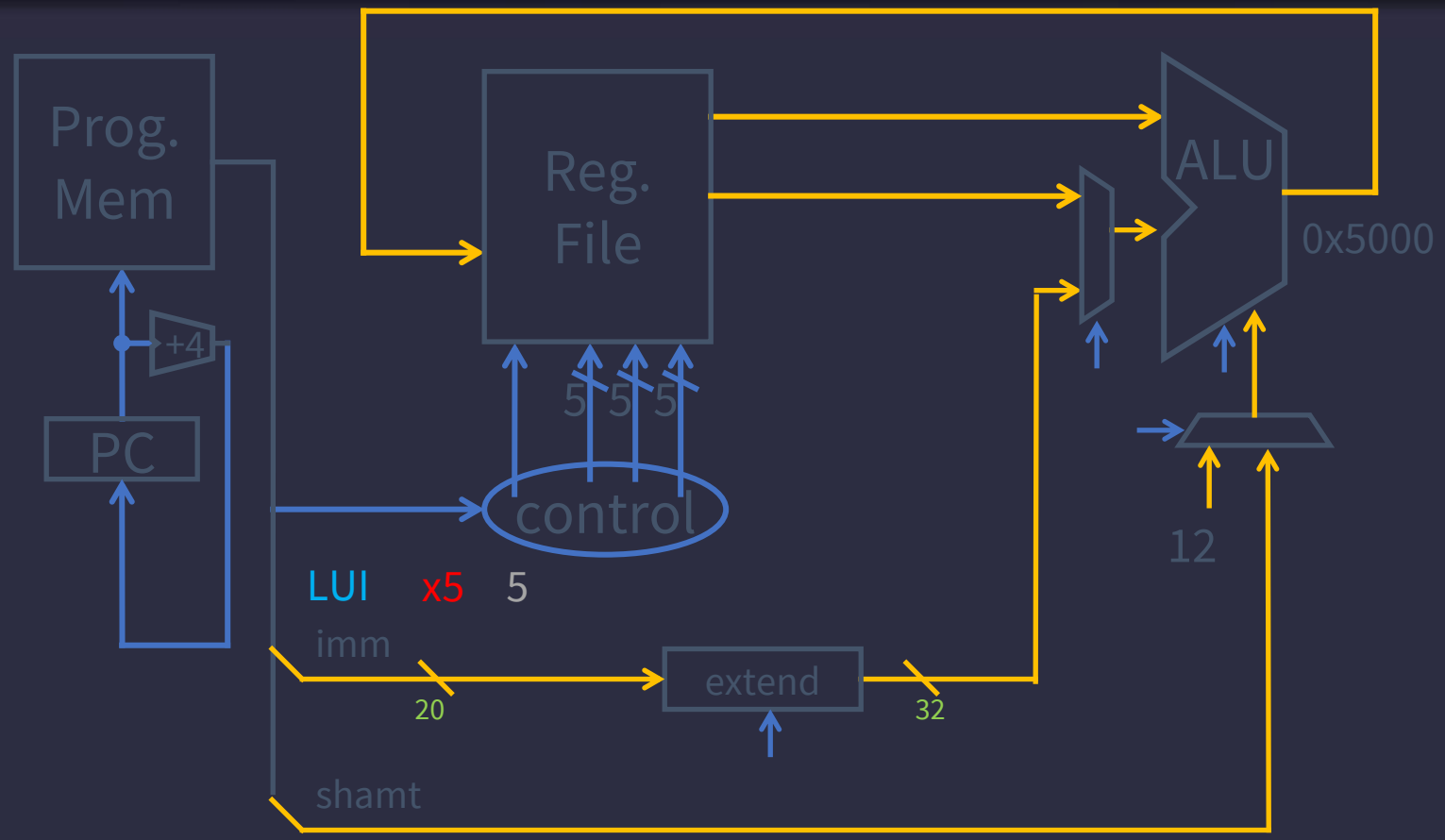


U-Type (1): Load Upper Immediate



op	mnemonic	description
0110111	LUI rd, imm	R[rd] = sign_ext(imm) << 12

Load Upper Immediate



Fetch | Decode | Execute | Memory | WB

skip

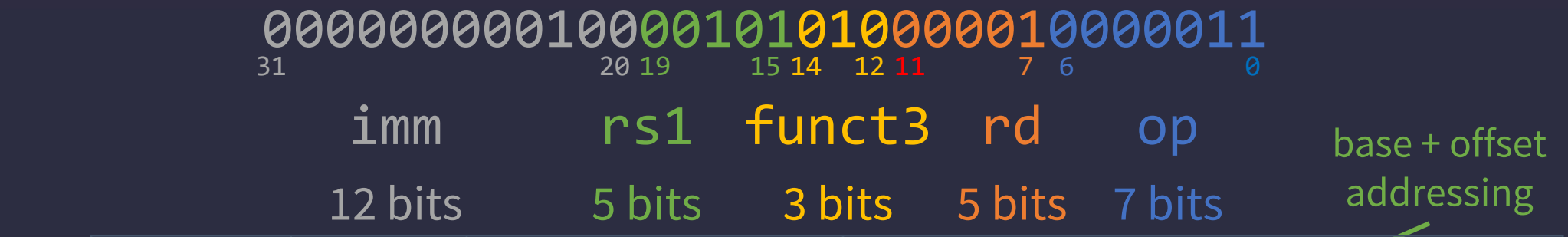
Example: x5 = 0x5000

LUI x5, 5

RISC-V Instruction Types

- Arithmetic/Logical
  - R-type: result and two source registers, shift amount
  - I-type: result and source register, shift amount in 16-bit immediate with sign/zero extension
  - U-type: result register, 16-bit immediate with sign/zero extension
- Memory Access
  - I-type for loads and S-type for stores
  - load/store between registers and memory
  - word, half-word and byte operations
- Control flow
  - U-type: jump-and-link
  - I-type: jump-and-link register
  - SB-type: conditional branches: pc-relative addresses

I-Type (2): Load Instructions



base + offset addressing

op	funct3	mnemonic	Description
0000011	000	LB rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	001	LH rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	010	LW rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	011	LD rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	100	LBU rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	101	LHU rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	110	LWU rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]

signed offsets

I-Type (2): Load Instructions

```
00000000010000101010000010000011
31                  20 19   15 14 12 11    7 6         0
```

imm	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

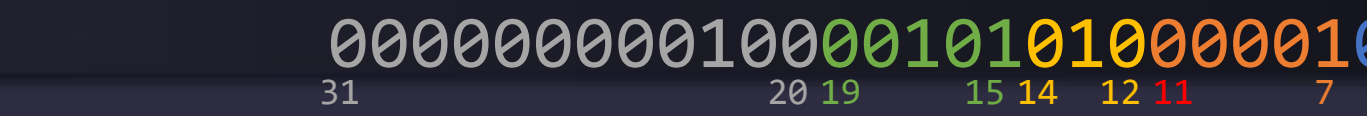
base + offset addressing

op	funct3	mnemonic	Description
0000011	000	LB rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	001	LH rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	010	LW rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	011	LD rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	100	LBU rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	101	LHU rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	110	LWU rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]

Example: x1 = Mem[4+x5] # LW x1, x5, 4 LW x1 4(x5)

signed offset

I-Type (2): Load Instructions



base + offset addressing

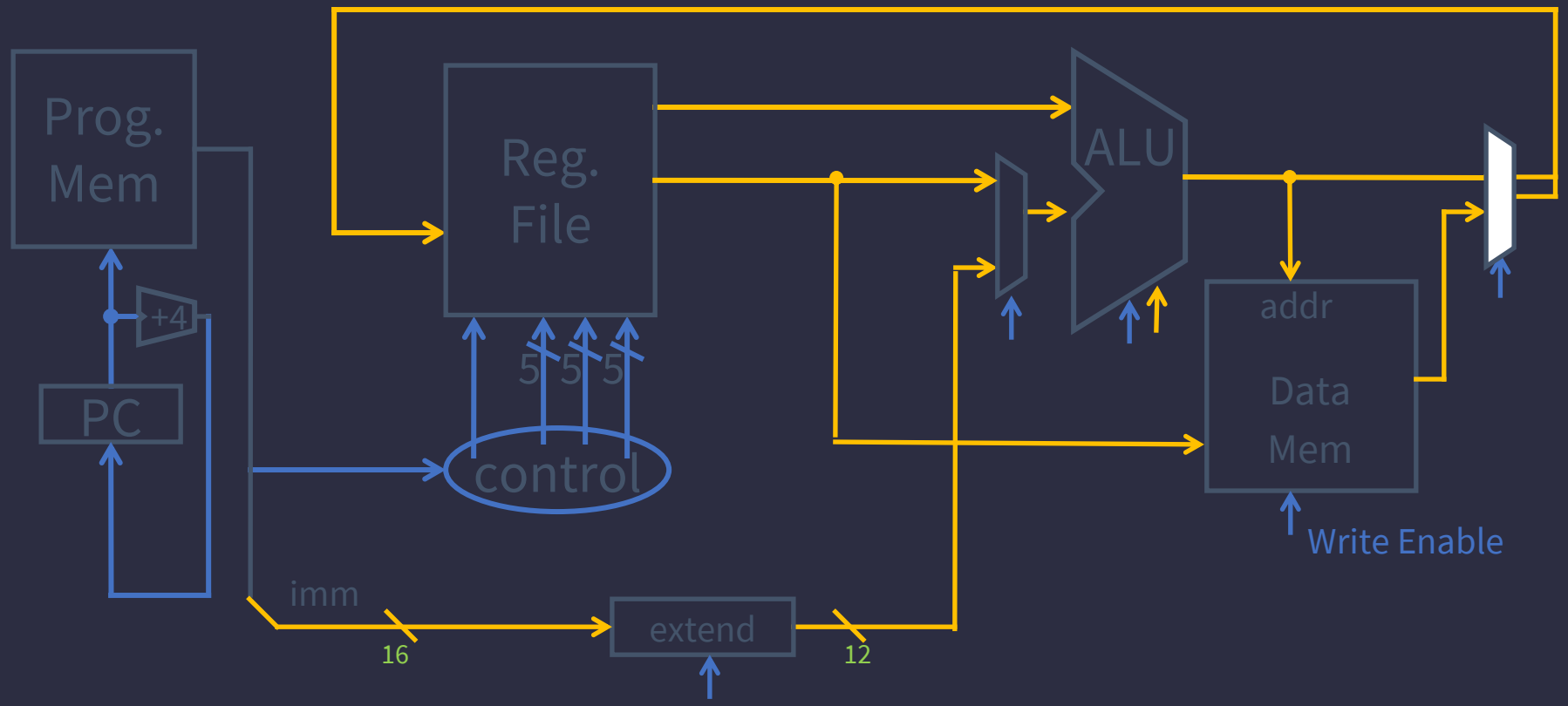
op	funct3	mnemonic	Description
0000011	000	LB rd, rs1, imm	R[rd] = sign_ext(Mem[imm+R[rs1]])
0000011	001	LH rd, rs1, imm	R[rd] = sign_ext(Mem[imm+R[rs1]])
0000011	010	LW rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	011	LD rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	100	LBU rd, rs1, imm	R[rd] = zero_ext(Mem[imm+R[rs1]])
0000011	101	LHU rd, rs1, imm	R[rd] = zero_ext(Mem[imm+R[rs1]])
0000011	110	LWU rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]

Example: x1 = Mem[4+x5] # LW x1, x5, 4

signed offsets

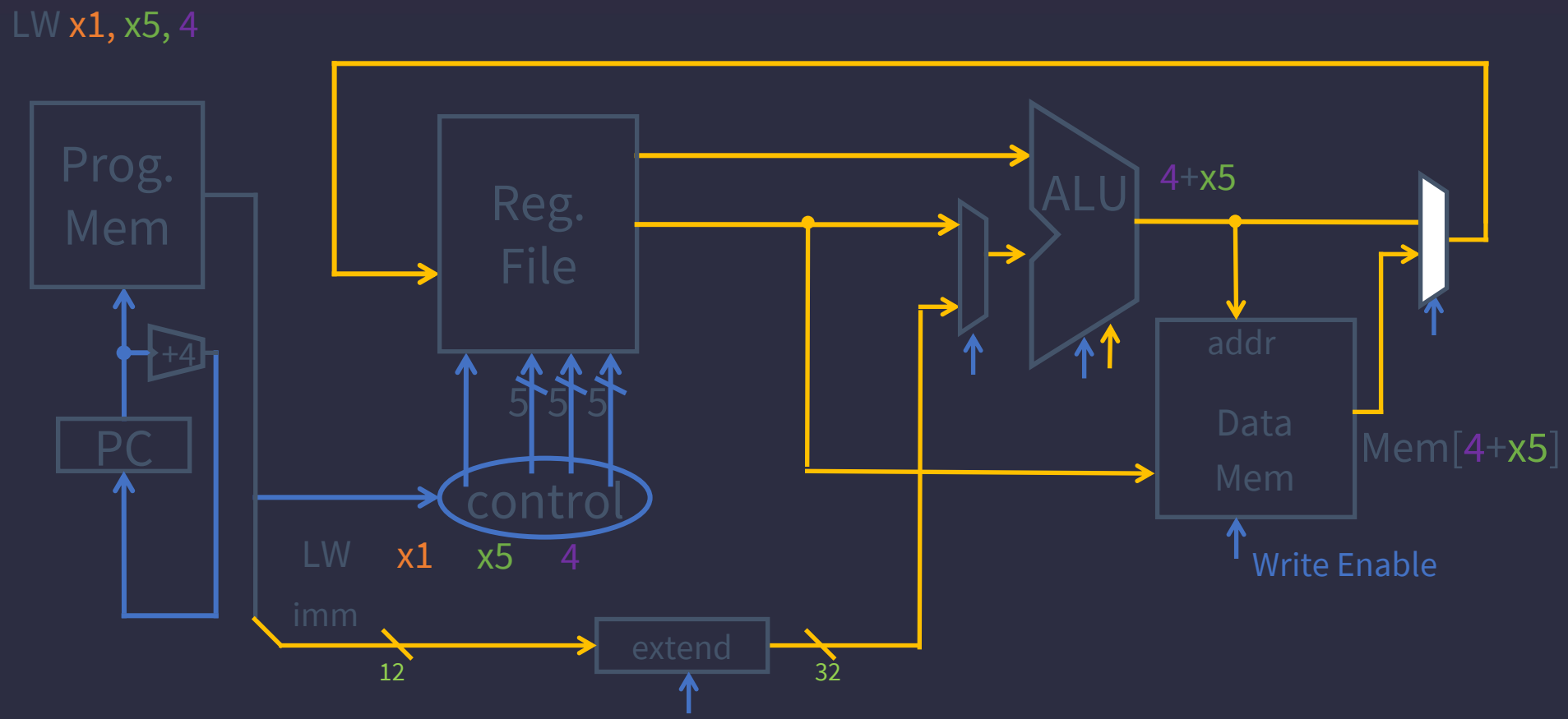
LW x1 4(x5)

Memory Operations: Load



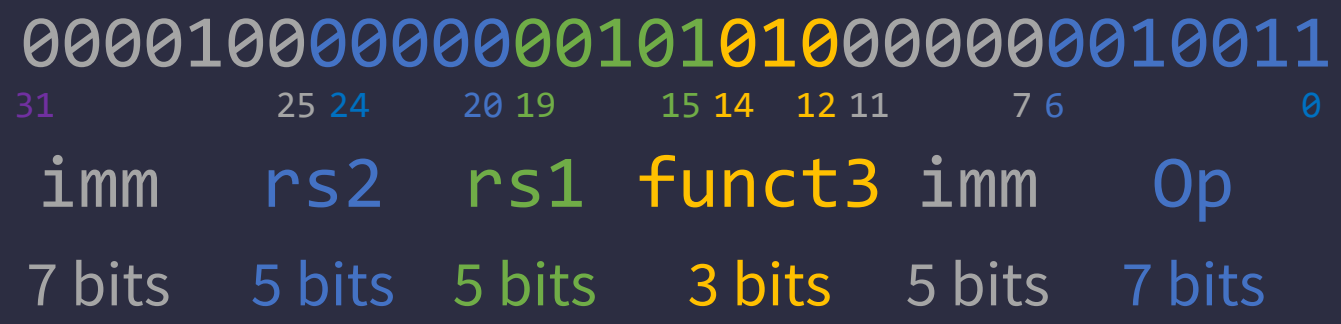
Example: x1 = Mem[4+x5] # LW x1, x5, 4 LW x1 4(x5)

Memory Operations: Load



Example: x1 = Mem[4+x5] # LW x1, x5, 4 LW x1 4(x5)

S-Type (1): Store Instructions



base + offset addressing

op	funct3	mnemonic	description
0100011	000	SB rs2, rs1, imm	Mem[sign_ext(imm)+R[rs1]] = R[rd]
0100011	001	SH rs2, rs1, imm	Mem[sign_ext(imm)+R[rs1]] = R[rd]
0100011	010	SW rs2, rs1, imm	Mem[sign_ext(imm)+R[rs1]] = R[rd]

signed offsets

S-Type (1): Store Instructions

```
0000100000000010101000000010011
31          25 24    20 19    15 14  12 11     7 6          0
```

imm	rs2	rs1	funct3	imm	Op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

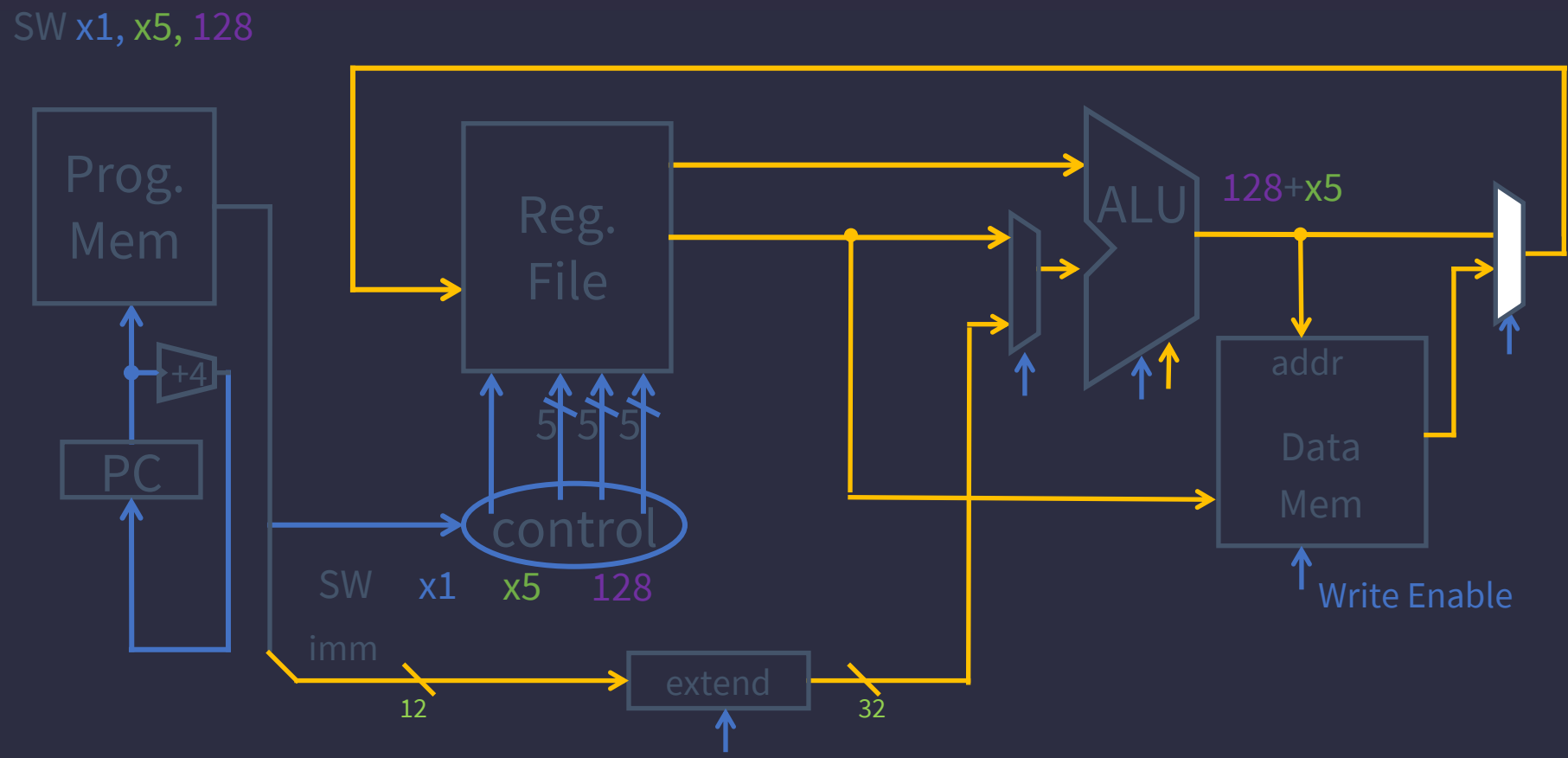
base + offset addressing

op	funct3	mnemonic	description
0100011	000	SB rs2, rs1, imm	Mem[sign_ext(imm)+R[rs1]] = R[rd]
0100011	001	SH rs2, rs1, imm	Mem[sign_ext(imm)+R[rs1]] = R[rd]
0100011	010	SW rs2, rs1, imm	Mem[sign_ext(imm)+R[rs1]] = R[rd]

signed offsets

Example: Mem[128+x5] = x1 # SW x1, x5, 128 SW x1 128(x5)

Memory Operations: Load



Example: Mem[4+x5] = x1 # SW x1, x5, 128 SW x1 128(x5)

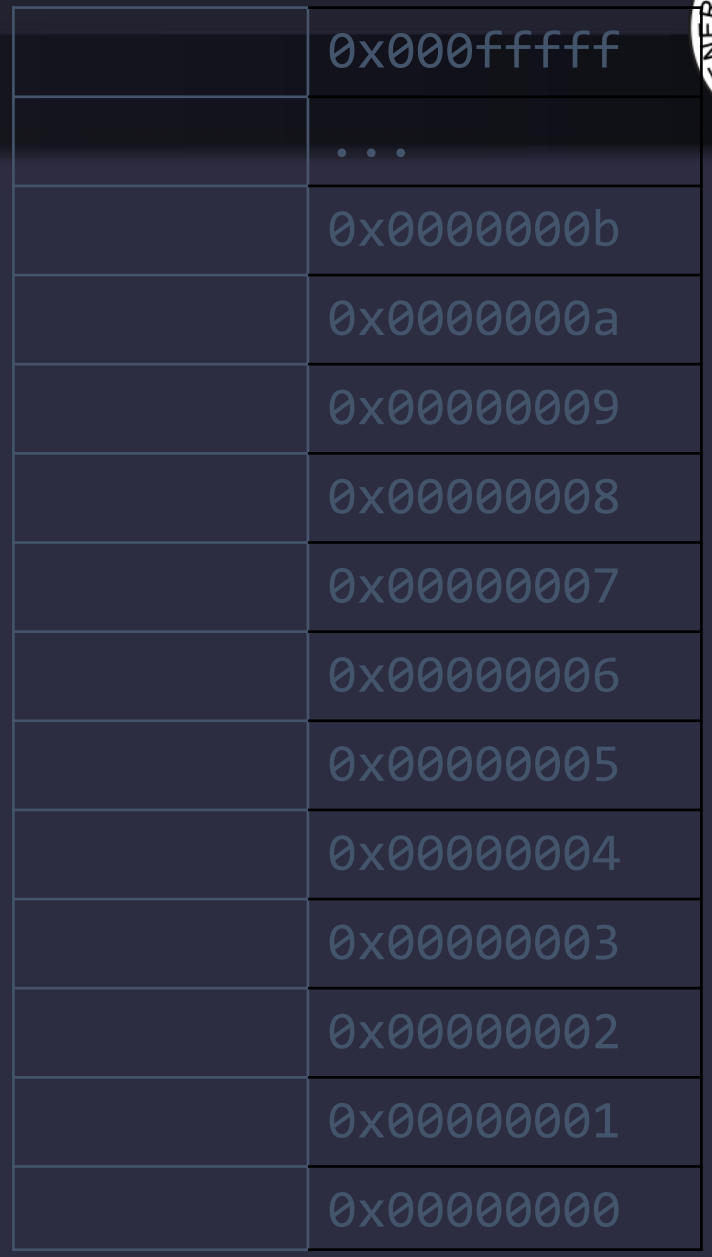
Memory Layout Options

- # x5 contains 5 (0x00000005)

- SB x5, x0, 0
- SB x5, x0, 2
- SW x5, x0, 8

- Two ways to store a word in memory.

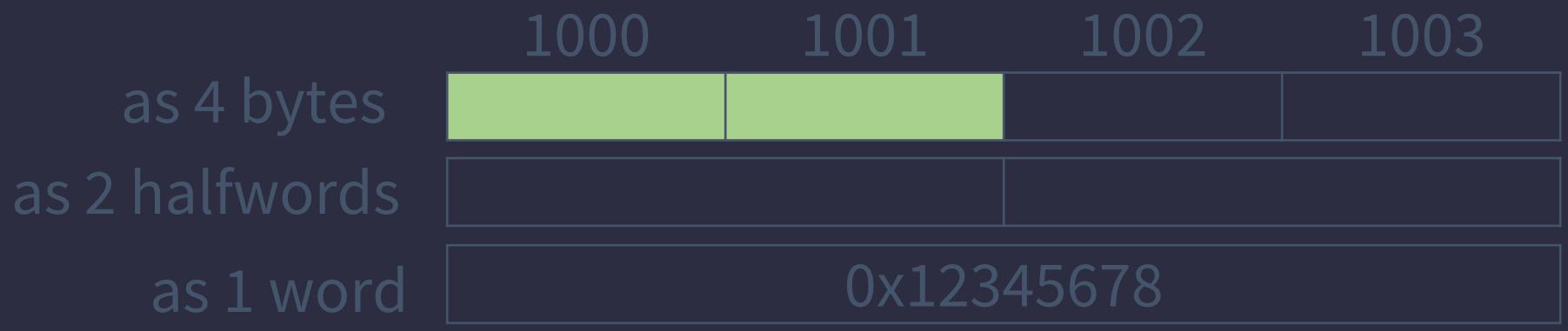
Endianness: ordering of bytes within a memory word



Little Endian

Endianness: Ordering of bytes within a memory word

Little Endian = least significant part first (RISC-V, x86)



0x78, 0x56 values will go in the byte-sized boxes with addresses 1000 and 1001

RISC-V Instruction Types

- Arithmetic/Logical
  - R-type: result and two source registers, shift amount
  - I-type: result and source register, shift amount in 16-bit immediate with sign/zero extension
  - U-type: result register, 16-bit immediate with sign/zero extension
- Memory Access
  - I-type for loads and S-type for stores
  - load/store between registers and memory
  - word, half-word and byte operations
- Control flow
  - U-type: jump-and-link
  - I-type: jump-and-link register
  - S-type: conditional branches: pc-relative addresses

UJ-Type (2): Jump and Link

op	Mnemonic	Description
1101111	JAL rd, imm	R[rd] = PC+4; PC=PC + sext(imm)

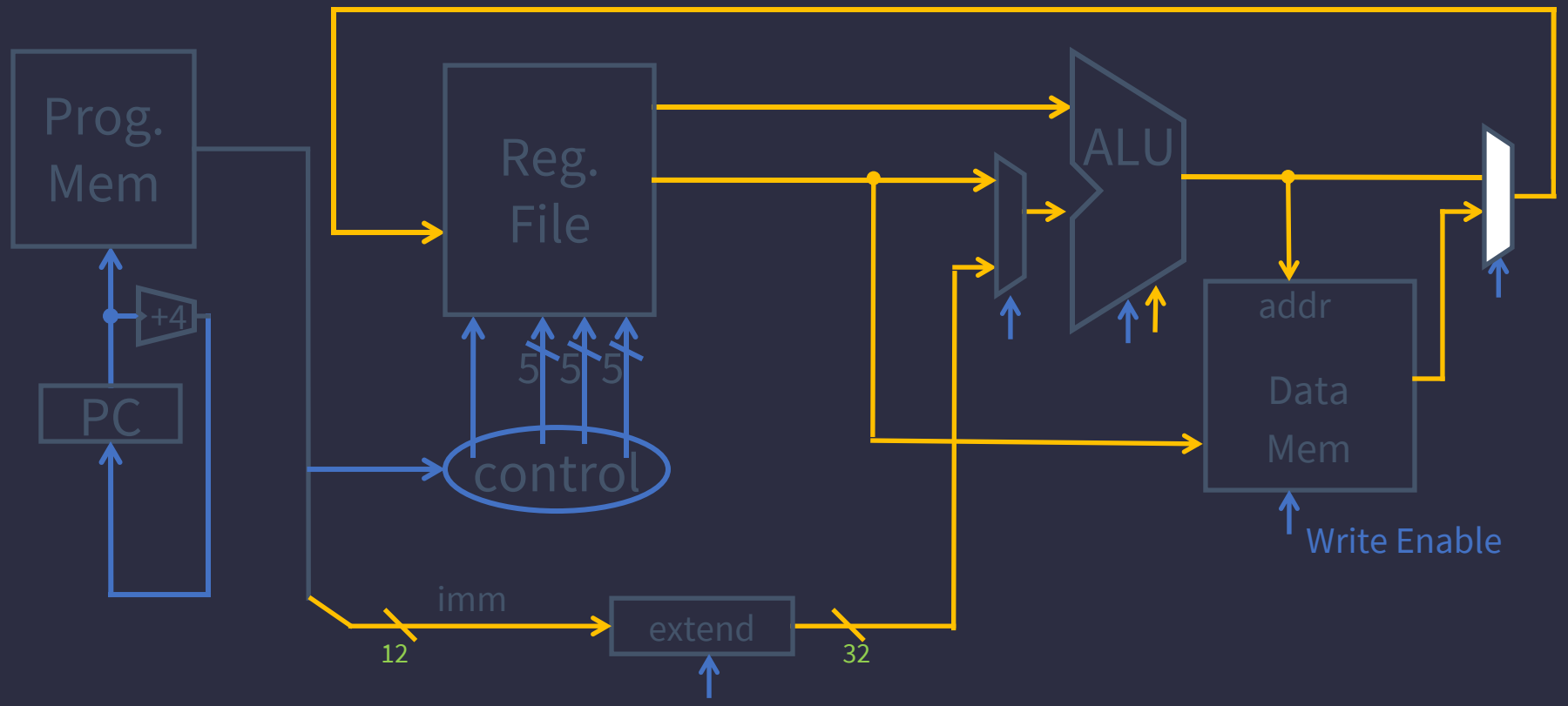
Example: x5 = PC+4 # JAL x5, 16

PC = PC + 16 (i.e. 16 == 8<<1)

Why?

Function/procedure calls

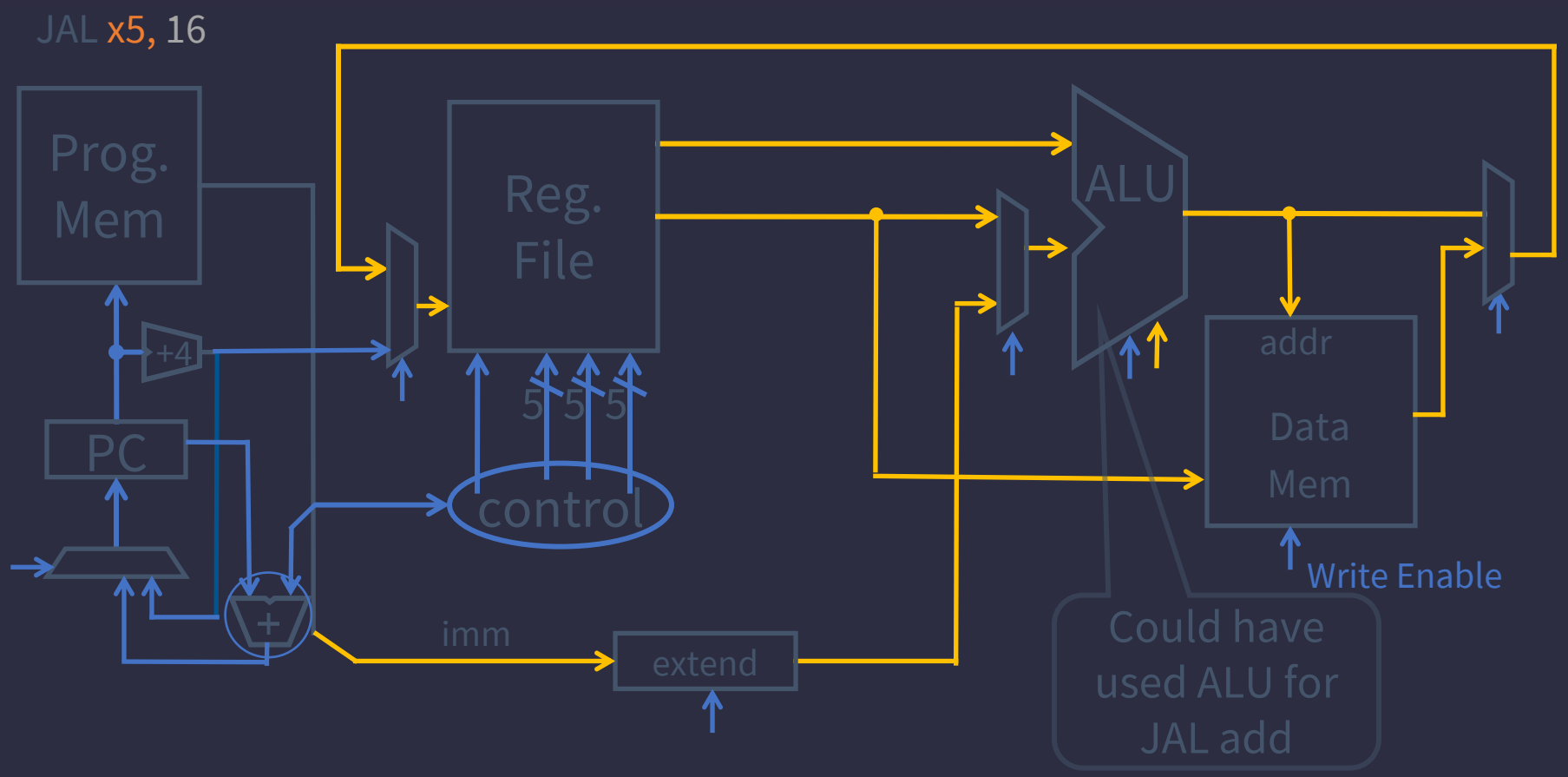
Jump and Link



Example: x5 = PC+4 # JAL x5, 16

PC = PC + 16 (i.e. 16 == 8<<1)

Jump and Link



Example: x5 = PC+4 # JAL x5, 16

PC = PC + 16 (i.e. 16 == 8<<1)

I-Type (3): Jump and Link Register

```
00000001000000100000000010111100111
31                  20 19    15 14  12 11     7 6          0
```

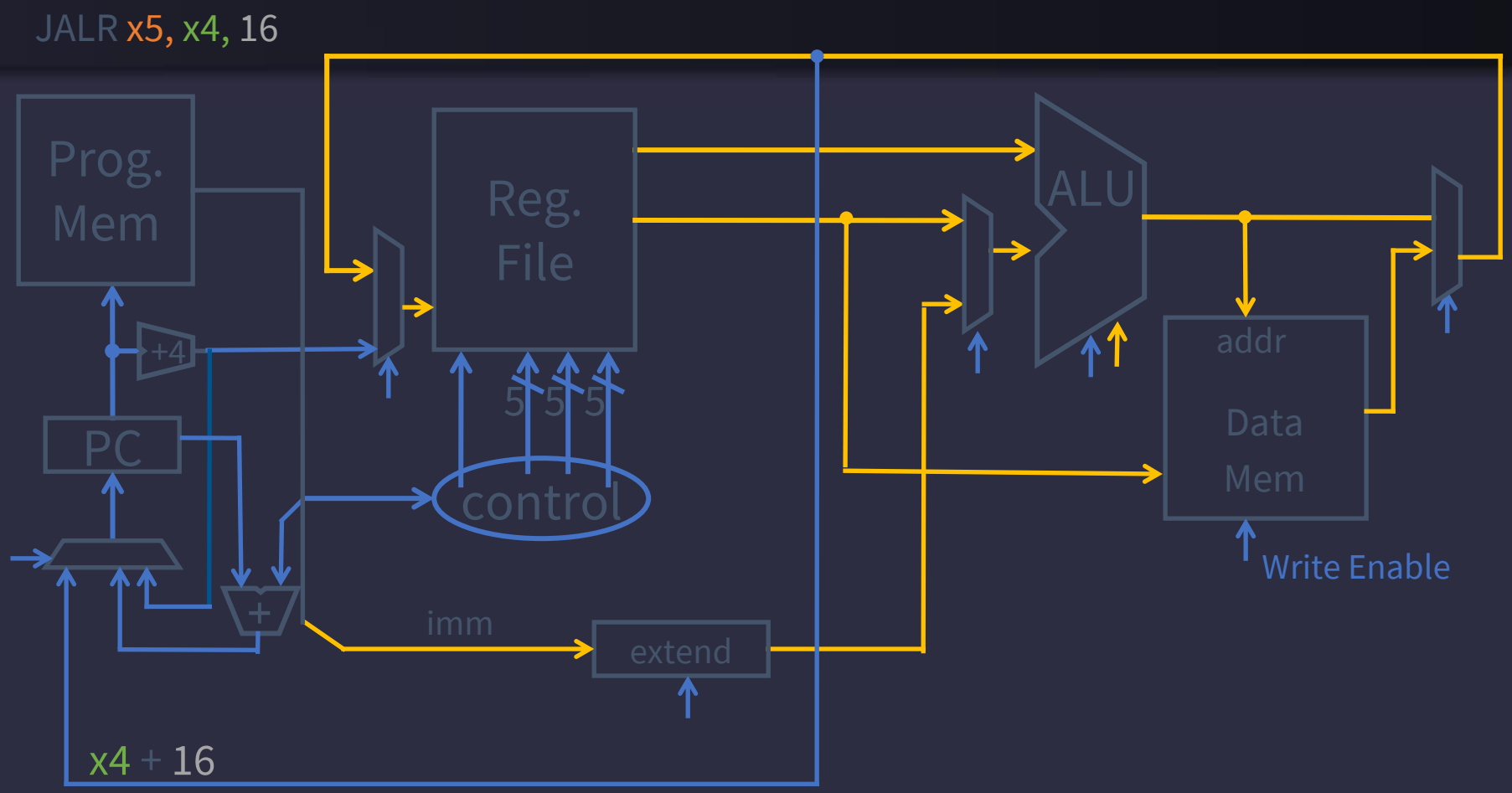
imm	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

op	funct3	Mnemonic	Description
1100111	000	JALR rd, rs1, imm	R[rd] = PC+4; PC=(R[rs1]+sign_ex(imm))&0xfffffffe

Example: x5 = PC+4 # JALR x5, x4, 16

PC = x4 + 16

Jump and Link Register



Example: x5 = PC+4 # JALR x5, x4, 16

PC = x4 + 16

Moving Beyond Jumps

- Can use Jump and link (JAL) or Jump and Link Register (JALR) instruction to jump to 0xabcd1234

What about a jump based on a condition?

- # assume 0 <= x3 <= 1
- if (x3 == 0) jump to 0xdecafe00

  else jump to 0xabcd1234

SB-Type (2): Branches

signed

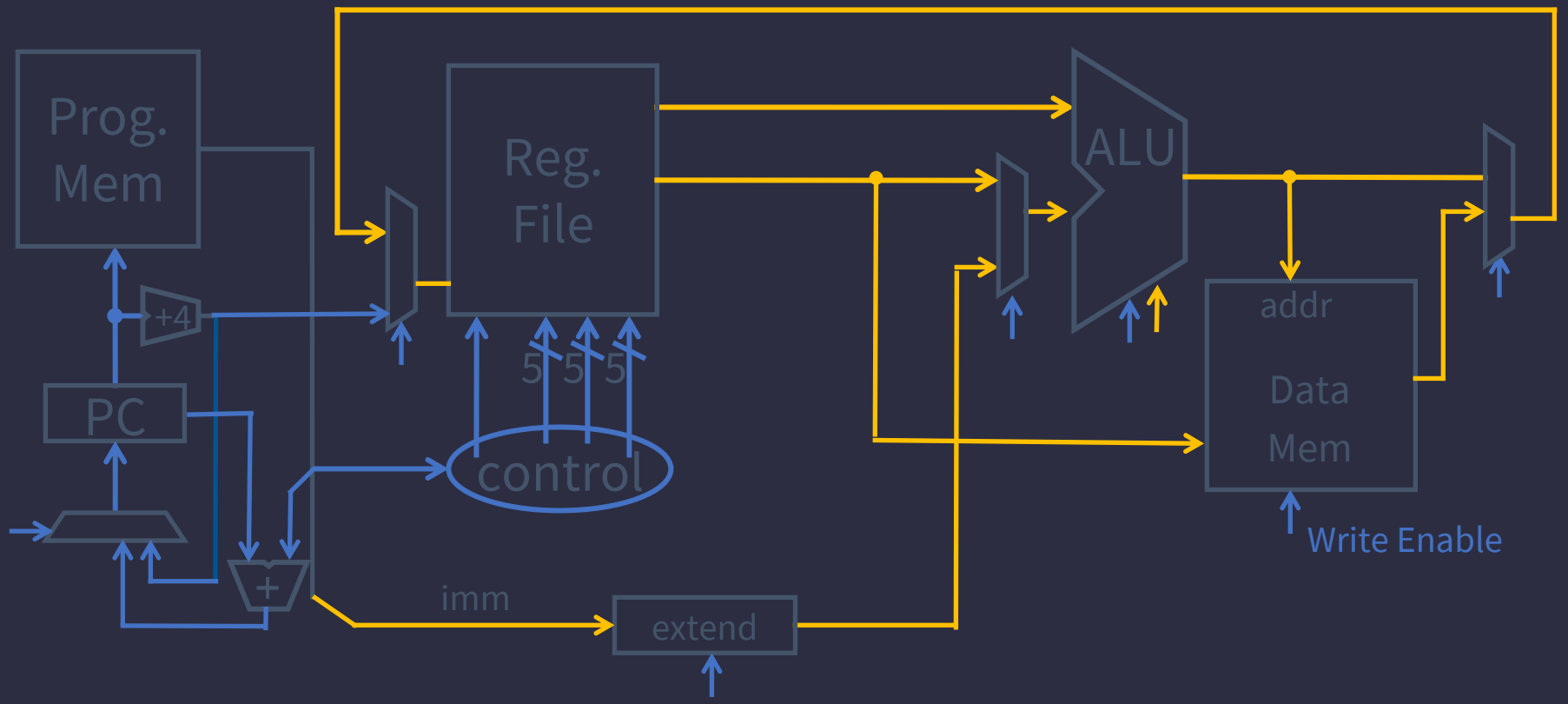
op	mnemonic	description
1100011	BEQ rs1, rs2, imm	PC=(R[rs1] == R[rs2] ? PC+sext(imm)<<1 : PC+4)
1100011	BNE rs1, rs2, imm	PC=(R[rs1] != R[rs2] ? PC+sext(imm)<<1 : PC+4)

Example: BEQ x5, x1, 128

```
if(R[x5]==R[x1])
      PC = PC + 128   (i.e. 128 == 64<<1)
```

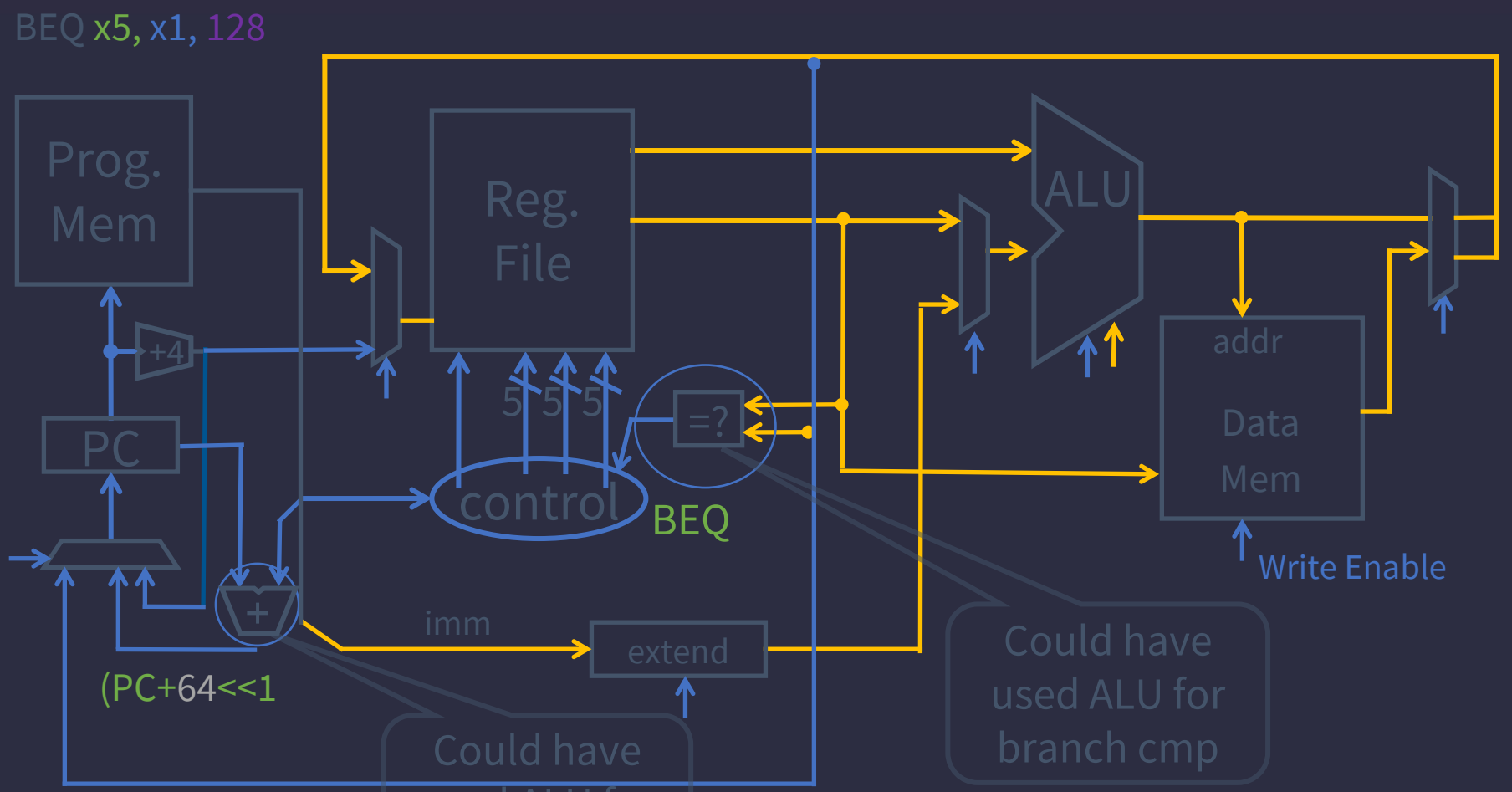
A word about all these +'s…

Control Flow: Branches



Example: BEQ x5, x1, 128

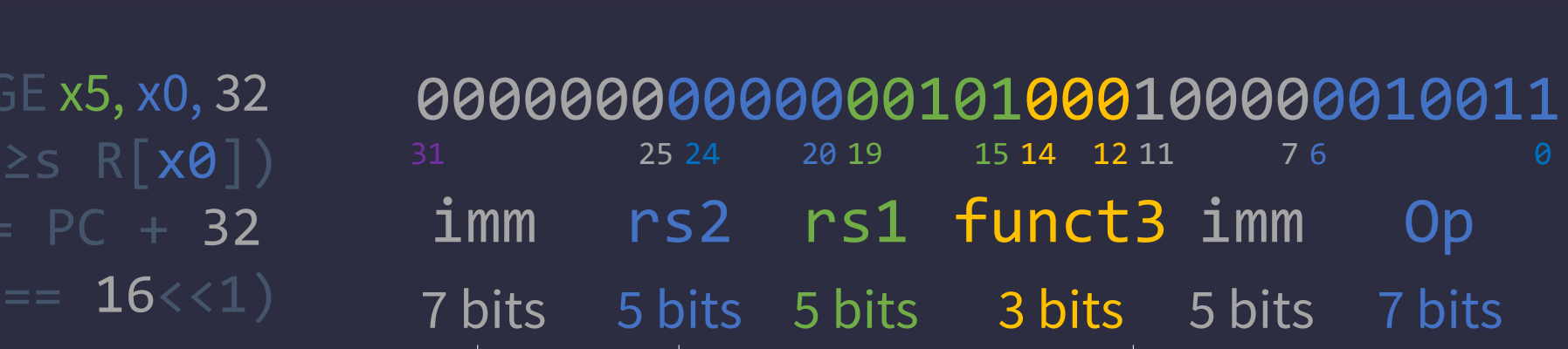
Control Flow: Branches



Example: BEQ x5, x1, 128

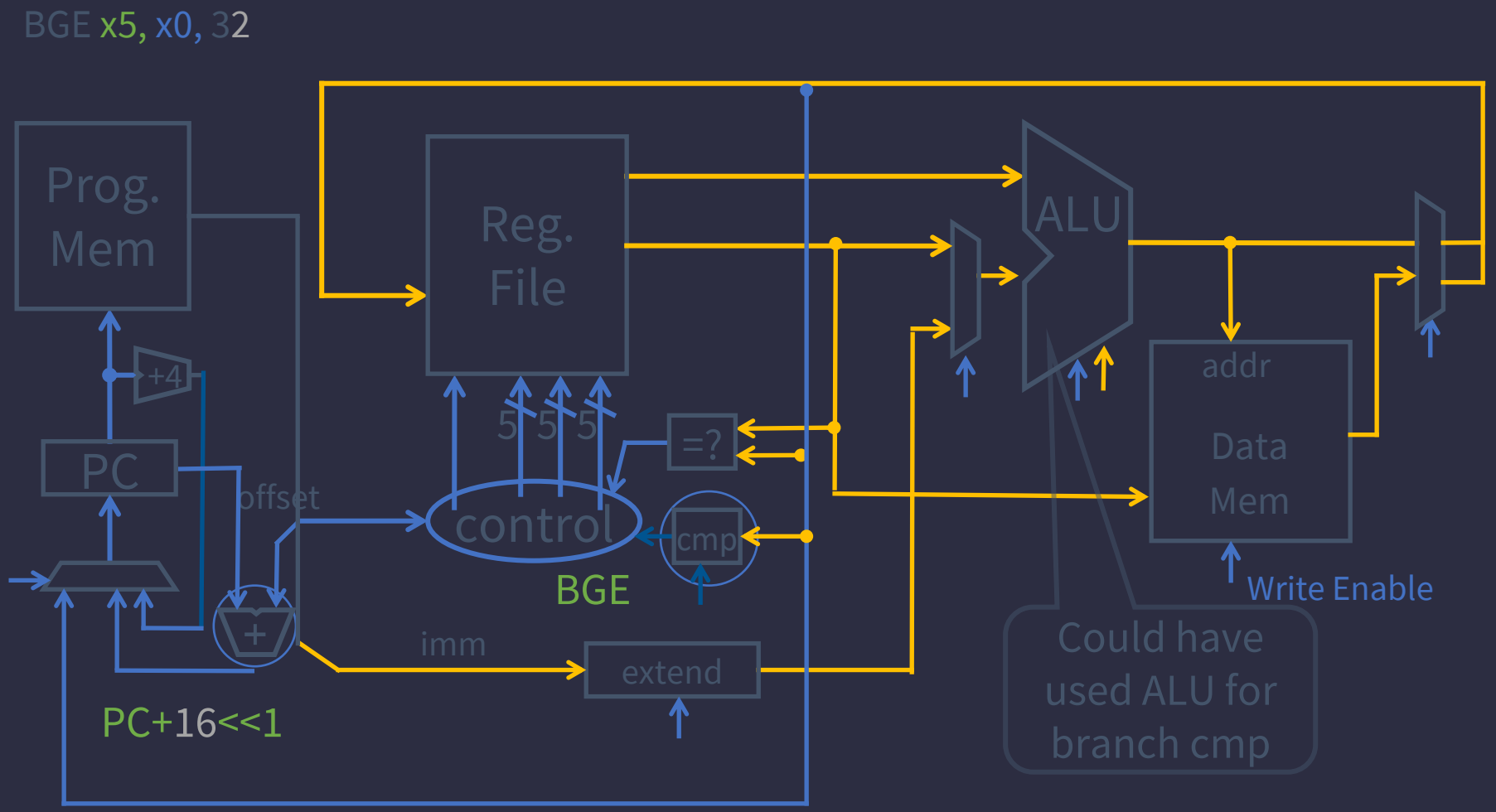
SB-Type (3): Conditional Jumps

Example: BGE x5, x0, 32
if(R[x5] ≥s R[x0])
PC = PC + 32
(i.e. 32 == 16<<1)



op	funct3	mnemonic	description
1100011	100	BLT rs1, rs2, imm	PC=(R[rs1] <s R[rs2] ? sext(imm)<<1 : PC+4) PC +
1100011	101	BGE rs1, rs2, imm	PC=(R[rs1] >=s R[rs2] ? sext(imm)<<1 : PC+4) PC +
1100011	110	BLTU rs1, rs2 imm	PC=(R[rs1] <u R[rs2] ? sext(imm)<<1 : PC+4) PC +
1100011	111	BGEU rs1, rs2, imm	PC=(R[rs1] >=u R[rs2] ? sext(imm)<<1 : PC+4) PC +

Control Flow: More Branches



Example: BGE x5, x0, 32

RISC-V Instruction Types

- Arithmetic/Logical
  - R-type: result and two source registers, shift amount
  - I-type: result and source register, shift amount in 16-bit immediate with sign/zero extension
  - U-type: result register, 16-bit immediate with sign/zero extension
- Memory Access
  - I-type for loads and S-type for stores
  - load/store between registers and memory
  - word, half-word and byte operations
- Control flow
  - U-type: jump-and-link
  - I-type: jump-and-link register
  - S-type: conditional branches: pc-relative addresses

Summary

We have all that it takes to build a processor!

- Arithmetic Logic Unit (ALU)
- Register File
- Memory

RISC-V processor and ISA is an example of a Reduced Instruction Set Computers (RISC)

- Simplicity is key, thus enabling us to build it!

We now know the data path for the MIPS ISA:

- register, memory and control instructions