Processor

Anne Bracy

CS 3410

Computer Science

Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, and Sirer.

See P&H Chapter: 2.16-2.20, 4.1-4.4, Appendix B

Goal for Today

Understanding the basics of a processor

We now have the technology to build a CPU!

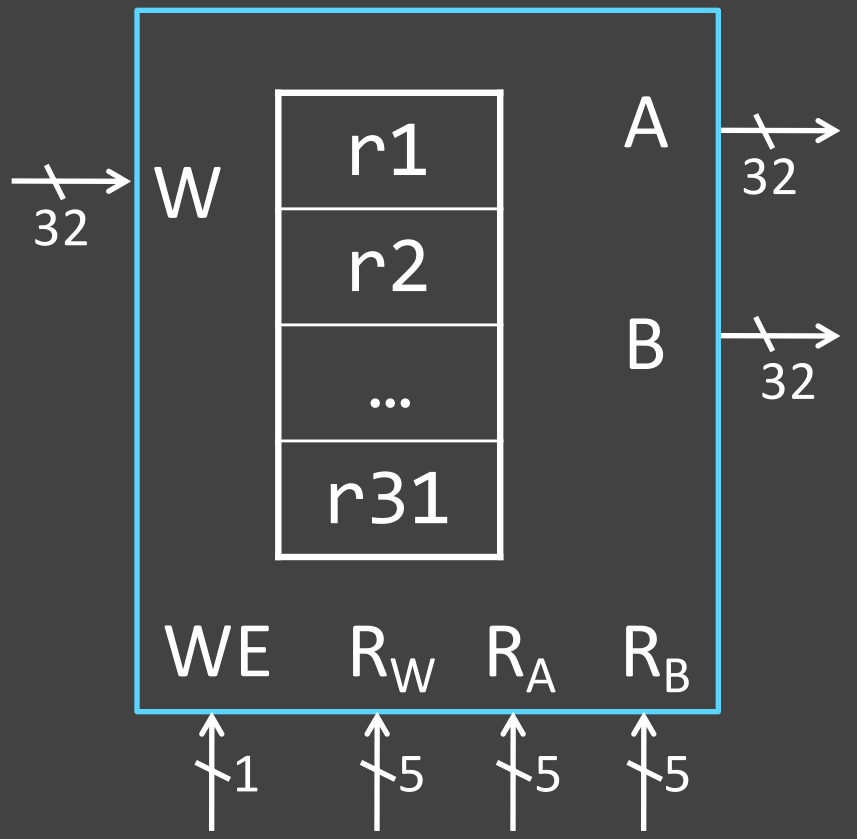
Putting it all together:

- Arithmetic Logic Unit (ALU)—Lab0 & 1, Lecture 2 & 3
- Register File—Lecture 4 and 5
- Memory—Lecture 5
  - SRAM: cache
  - DRAM: main memory
- MIPS Instructions & how they are executed

MIPS Register file

MIPS register file

- 32 x 32-bit registers
- r0 wired to zero
- Write port indexed via R_W
  - on falling edge when WE=1
- Read ports indexed via R_A, R_B

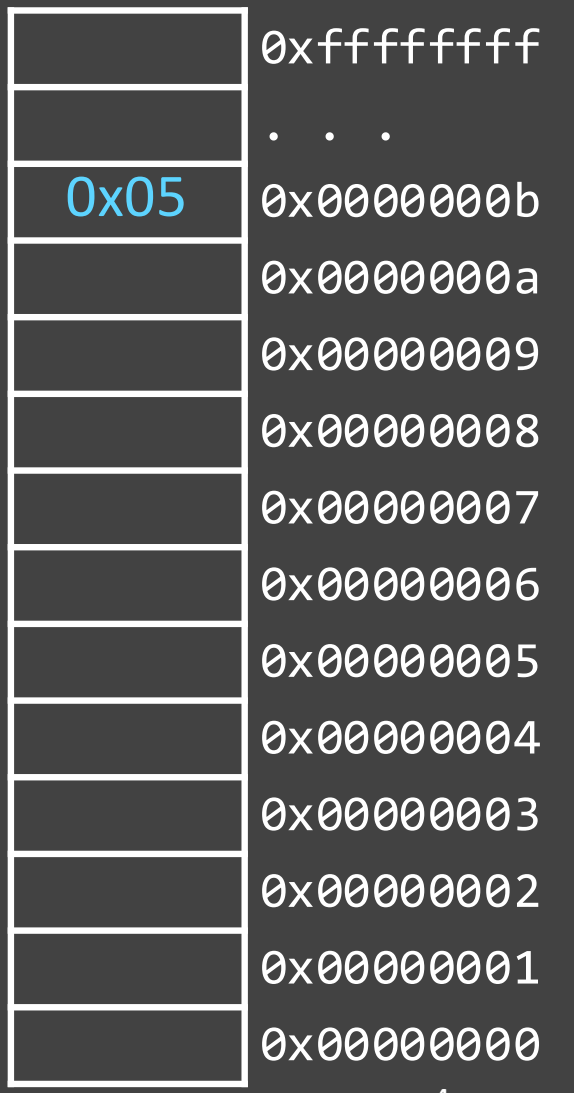


Registers

- Numbered from 0 to 31.
- Can be referred by number: $0, $1, $2, … $31
- Convention, each register also has a name:
  - $16 - $23 → $s0 - $s7, $8 - $15 → $t0 - $t7

[P&H p105]

MIPS Memory



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

  00: read word (4 byte aligned)

  01: write byte

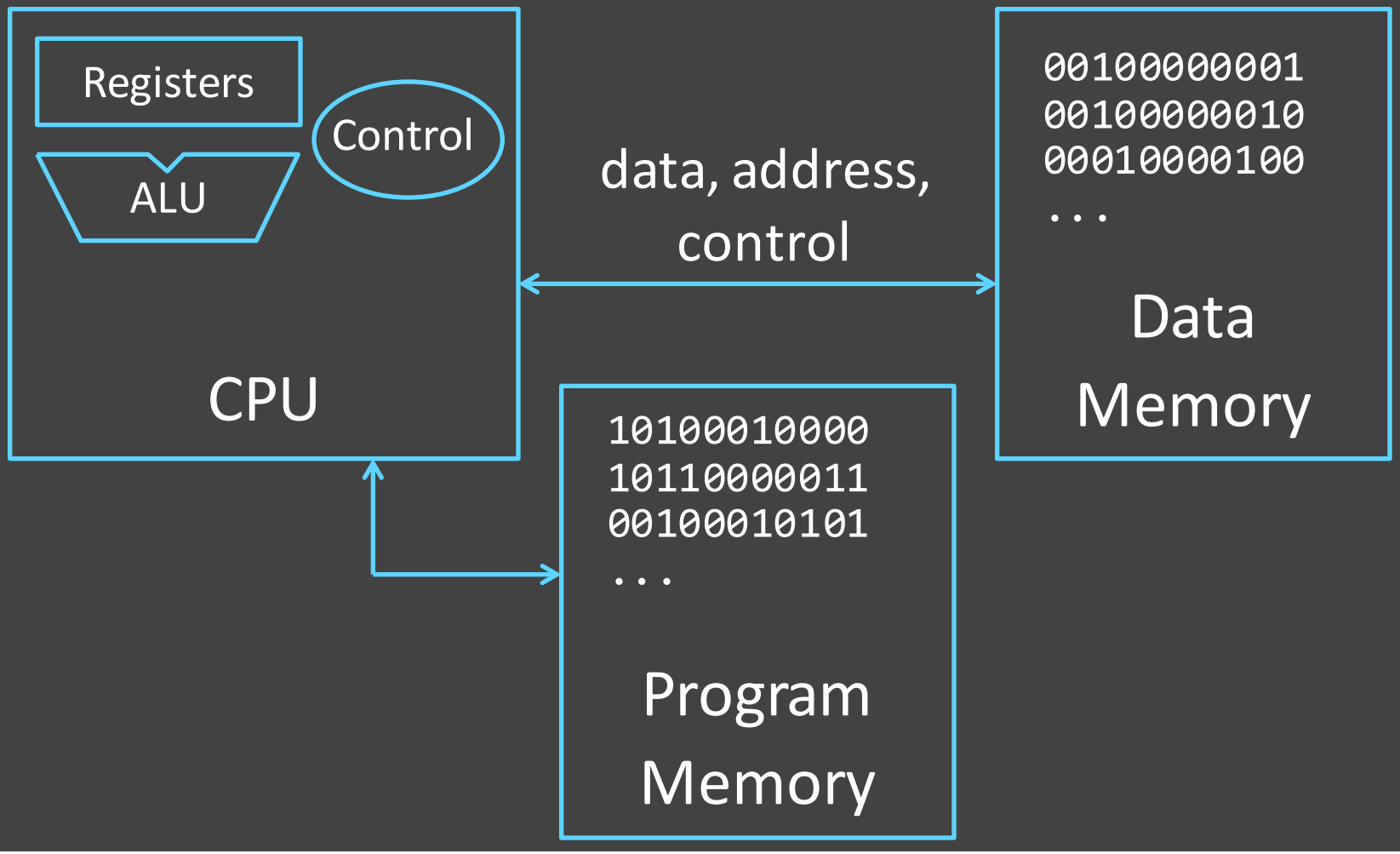
  10: write halfword (2 byte aligned)

  11: write word (4 byte aligned)

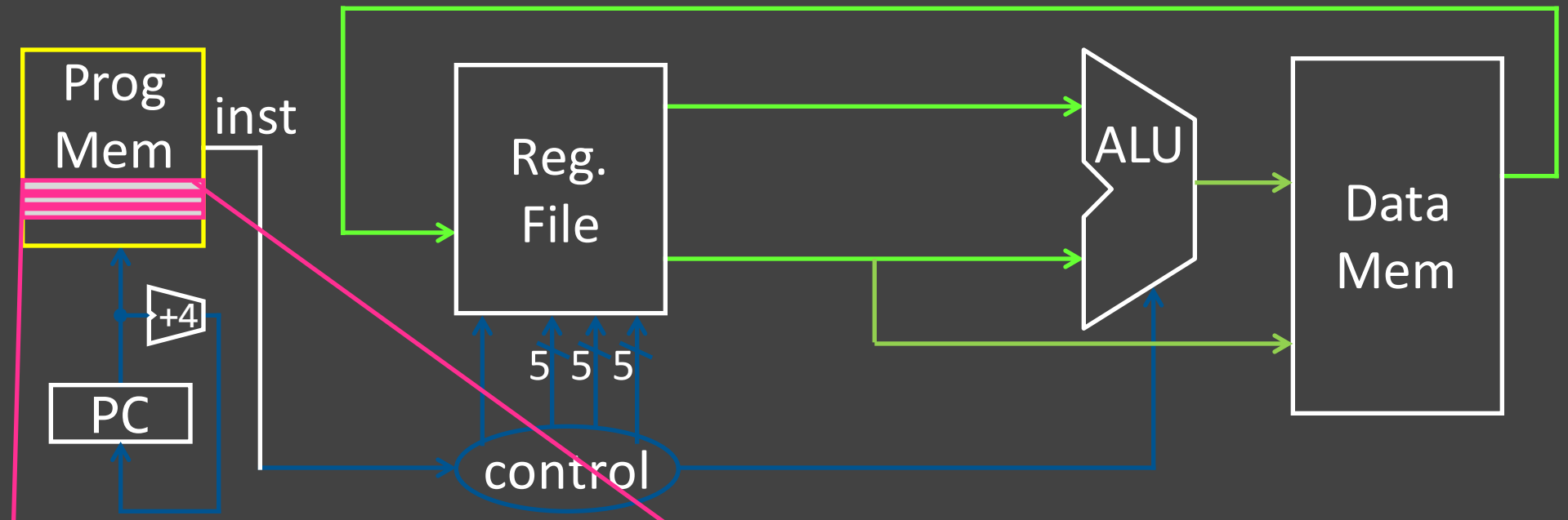
Basic Processor

A MIPS CPU with a (modified) Harvard architecture

- Modified: insns & data in common addr space
- Not von Neumann: ours access insn & data in parallel



Instruction Processing



Instructions:

stored in memory, encoded in binary

```
00100000000001000000000000001010
00100000000000100000000000000000
00000000001000100001100000101010
```

A basic processor

- fetches
- decodes
- executes

one instruction at a time

Levels of Interpretation: Instructions

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

```
main: addi r2, r0, 10
      addi r1, r0, 0
loop: slt r3, r1, r2
      ...
```

op=addi r0 r2 10

```
001000 00000 00010 0000000000001010
00100000000000010000000000000000
00000000001000100001100000101010
```

Instruction Set Architecture

ALU, Control, Register File, …

High Level Language

- C, Java, Python, Ruby, …
- Loops, control flow, variables

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

A Closer Look at 2 ISAs

MIPS (RISC) – ISA of 3410

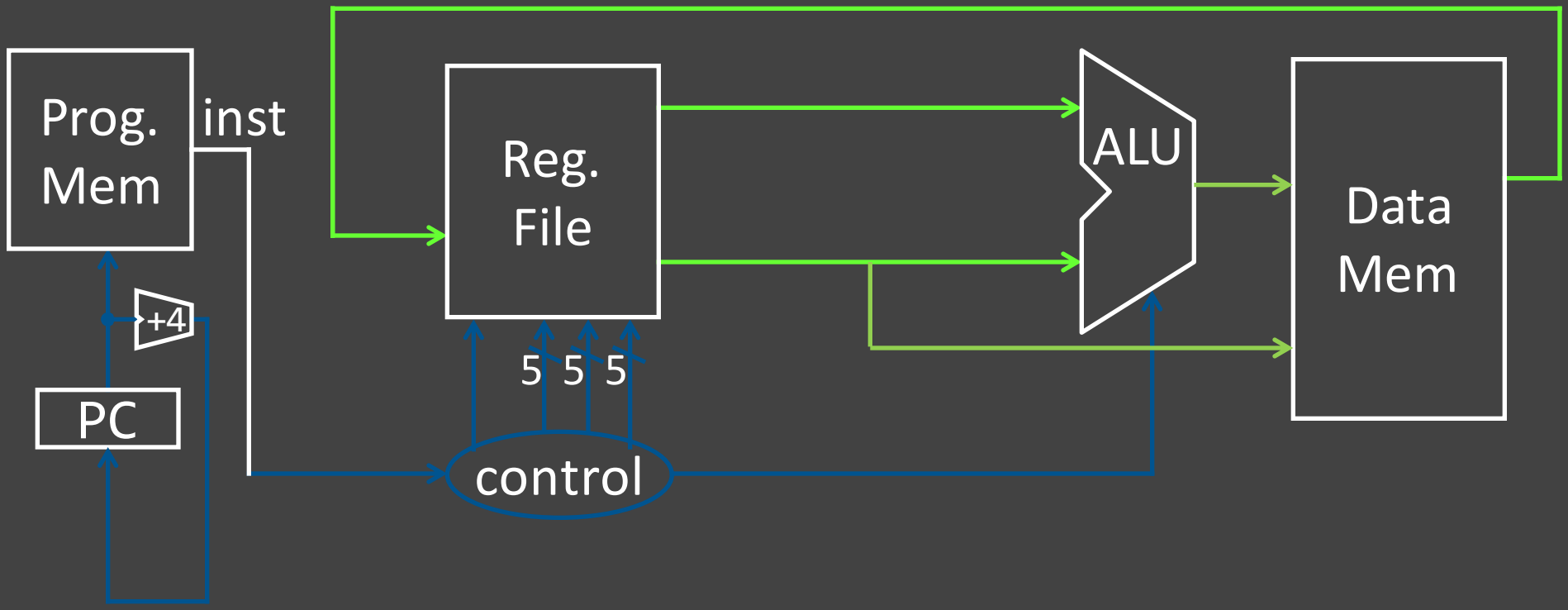
- ≈ 200 instructions, 32 bits each, 3 formats
  - mostly orthogonal
- all operands in registers
  - almost all are 32 bits each, can be used interchangeably
- ≈ 1 addressing mode: Mem[reg + imm] *"100 Main St."*

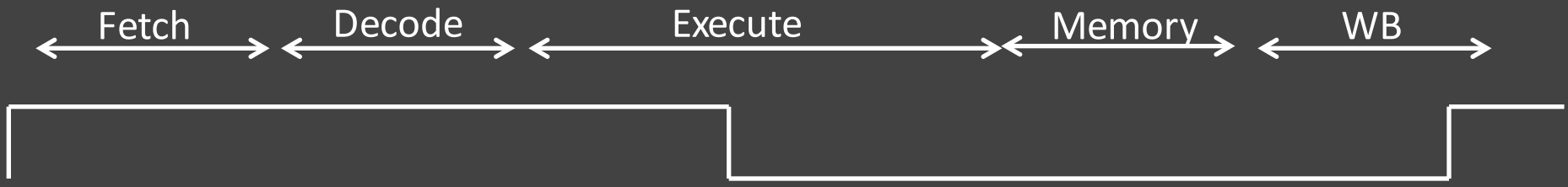
x86 (CISC) – ISA of your desktop & laptop

- > 1000 instructions, 1 to 15 bytes each
- operands in special registers, general purpose registers, memory, on stack, …
  - can be 1, 2, 4, 8 bytes, signed or unsigned
- 10s of addressing modes
  - e.g. Mem[segment + reg + reg*scale + offset]

"Blue house half a mile past the oak tree across from the gas station."

Five Stages of MIPS Datapath





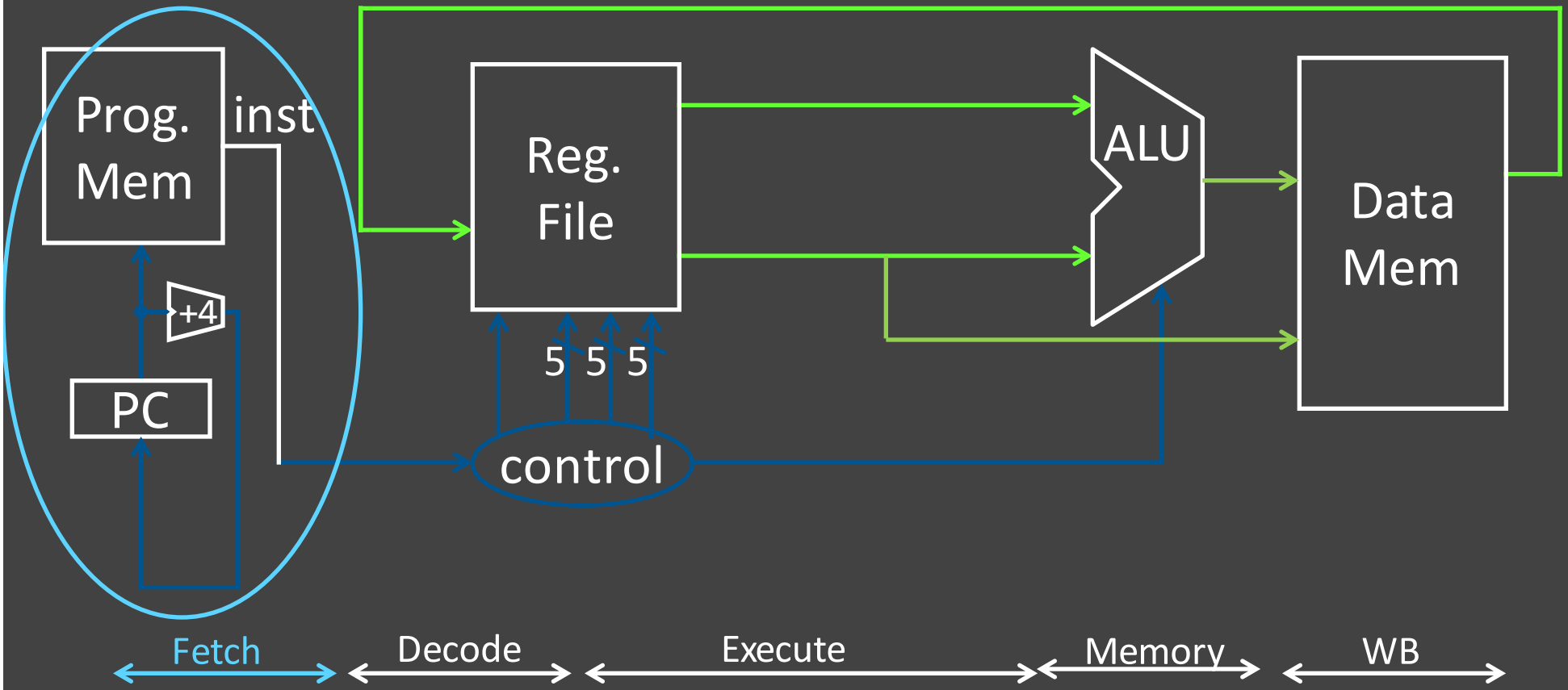
A Single cycle processor – this diagram is not 100% spatial

Five Stages of MIPS datapath

Basic CPU execution loop

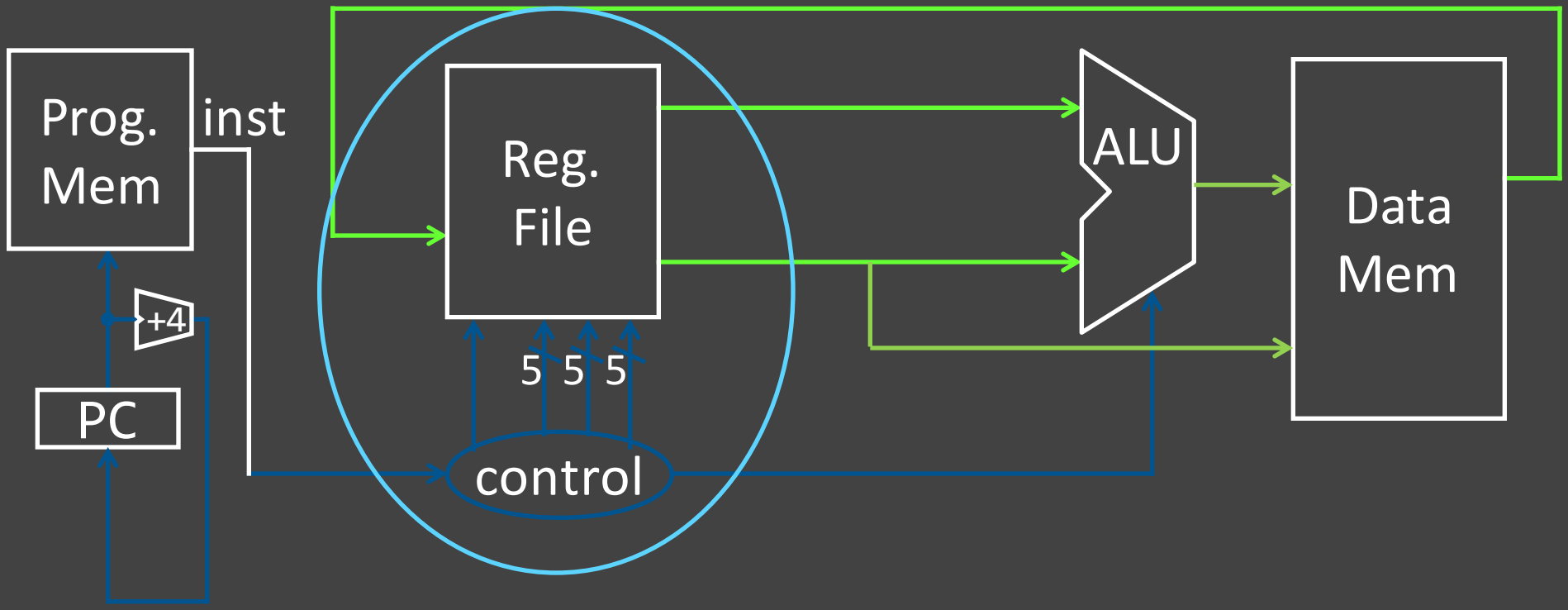
1. Instruction Fetch
2. Instruction Decode
3. Execution (ALU)
4. Memory Access
5. Register Writeback

Stage 1: Instruction Fetch



- Fetch 32-bit instruction from memory
- Increment PC = PC + 4

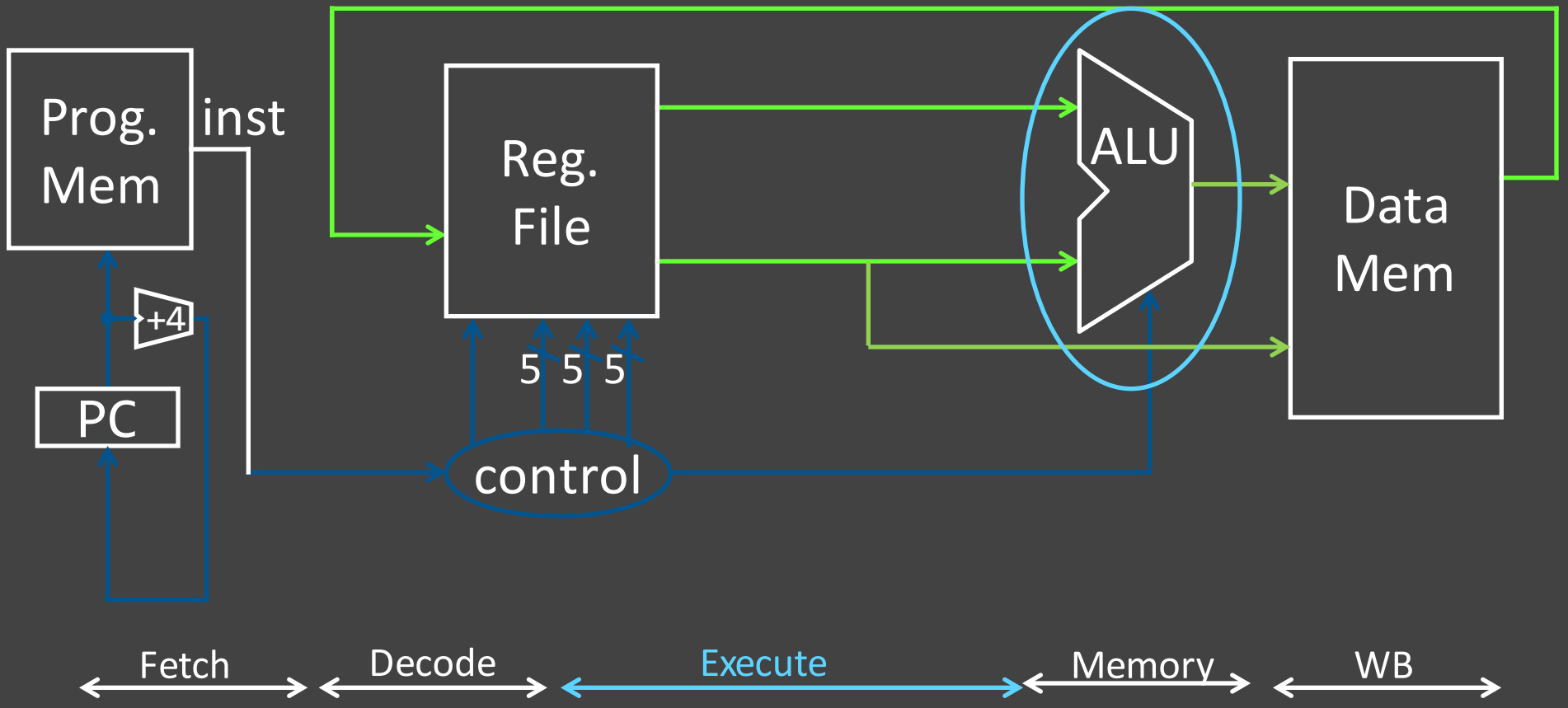
Stage 2: Instruction Decode



- Gather data from the instruction
- Read opcode; determine instruction type, field lengths
- Read in data from register file

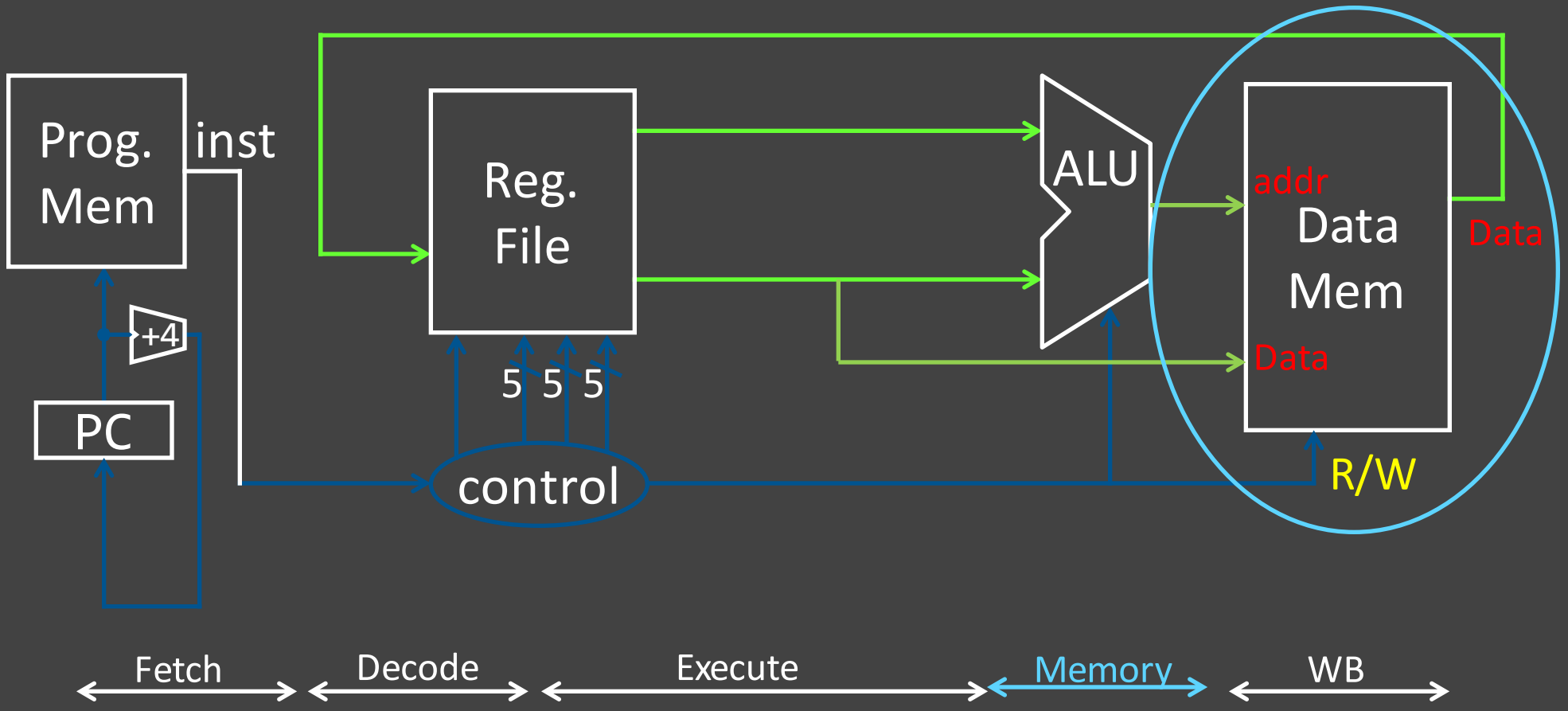
  (0, 1, or 2 reads for `jump`, `addi`, or `add`, respectively)

Stage 3: Execution (ALU)



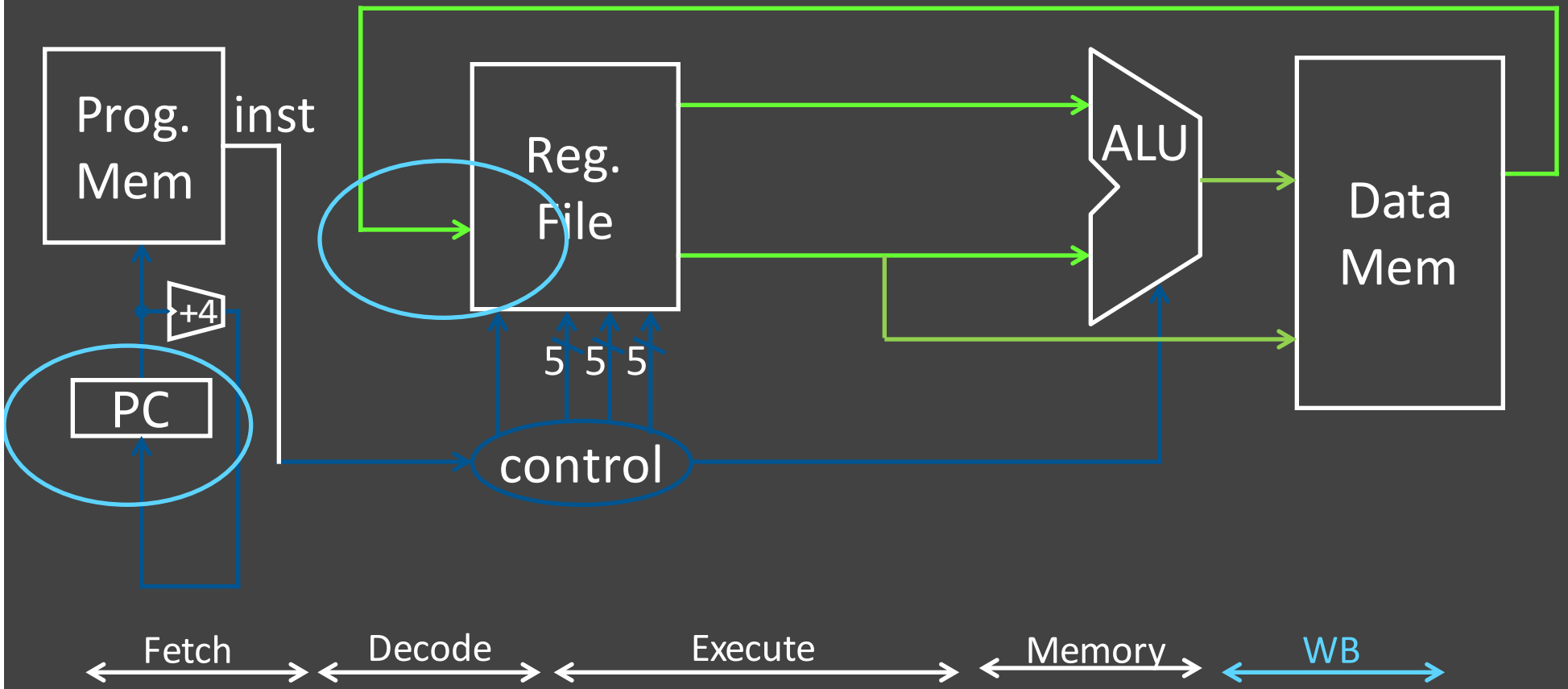
- Useful work done here (+, -, *, /), shift, logic operation, comparison (slt)
- Load/Store? lw $t2, 32($t3) → Compute address

Stage 4: Memory access



- Used by load and store instructions only
- Other 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

MIPS Instruction Types

Arithmetic/Logical

- R-type: result and two source registers, shift amount
- I-type: 16-bit immediate with sign/zero extension

Memory Access

- I-type
- load/store between registers and memory
- word, half-word and byte operations

Control flow

- J-type: fixed offset jumps, jump-and-link
- R-type: register absolute jumps
- I-type: conditional branches: pc-relative addresses

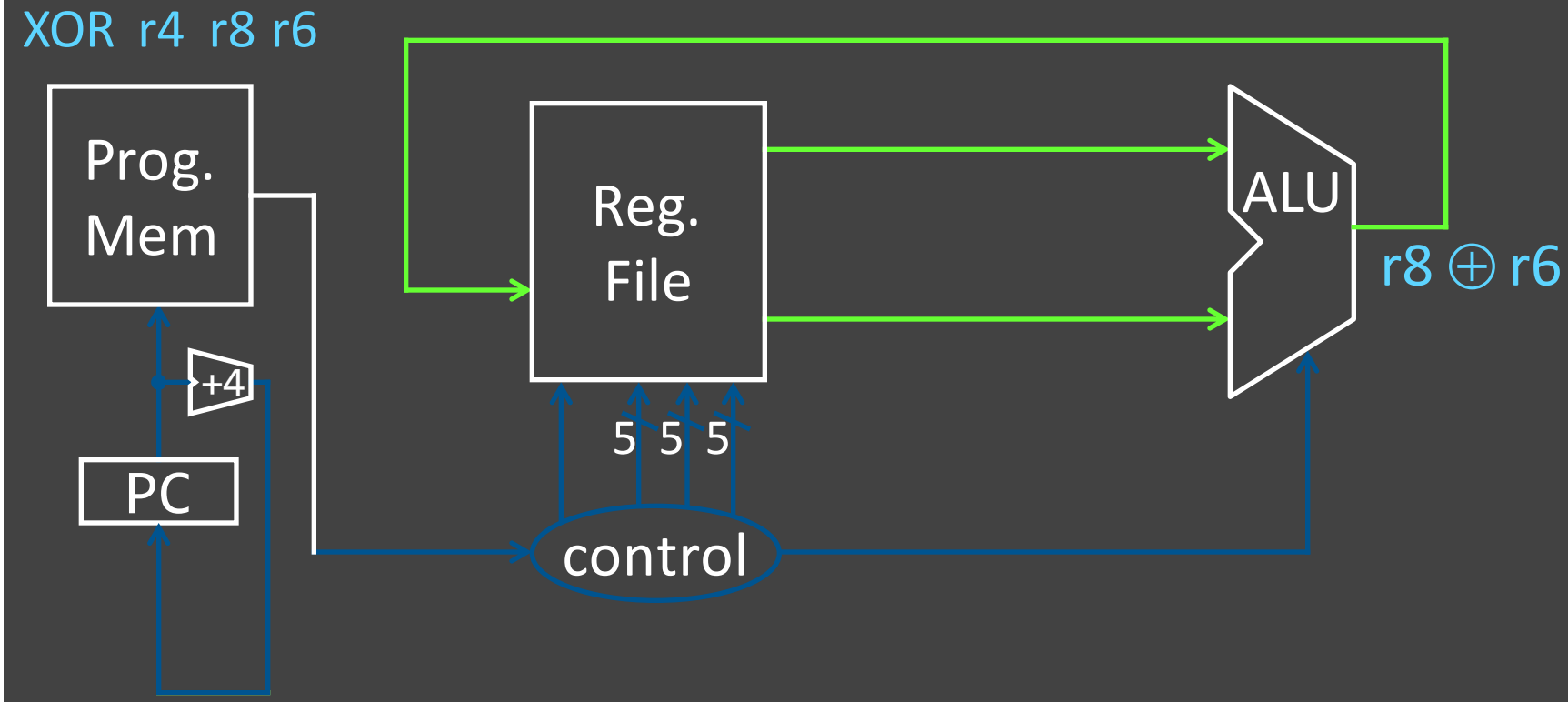
R-Type (1): Arithmetic and Logic

op	func	mnemonic	description
0x0	0x21	ADDU rd, rs, rt	R[rd] = R[rs] + R[rt]
0x0	0x23	SUBU rd, rs, rt	R[rd] = R[rs] – R[rt]
0x0	0x25	OR rd, rs, rt	R[rd] = R[rs] | R[rt]
0x0	0x26	XOR rd, rs, rt	R[rd] = R[rs] ⊕ R[rt]
0x0	0x27	NOR rd, rs rt	R[rd] = ~ (R[rs] | R[rt])

example: r4 = r8 ⊕ r6 # XOR r4, r8, r6

rd, rs, rt

Arithmetic and Logic



Example: r4 = r8 ⊕ r6 # XOR r4, r8, r6

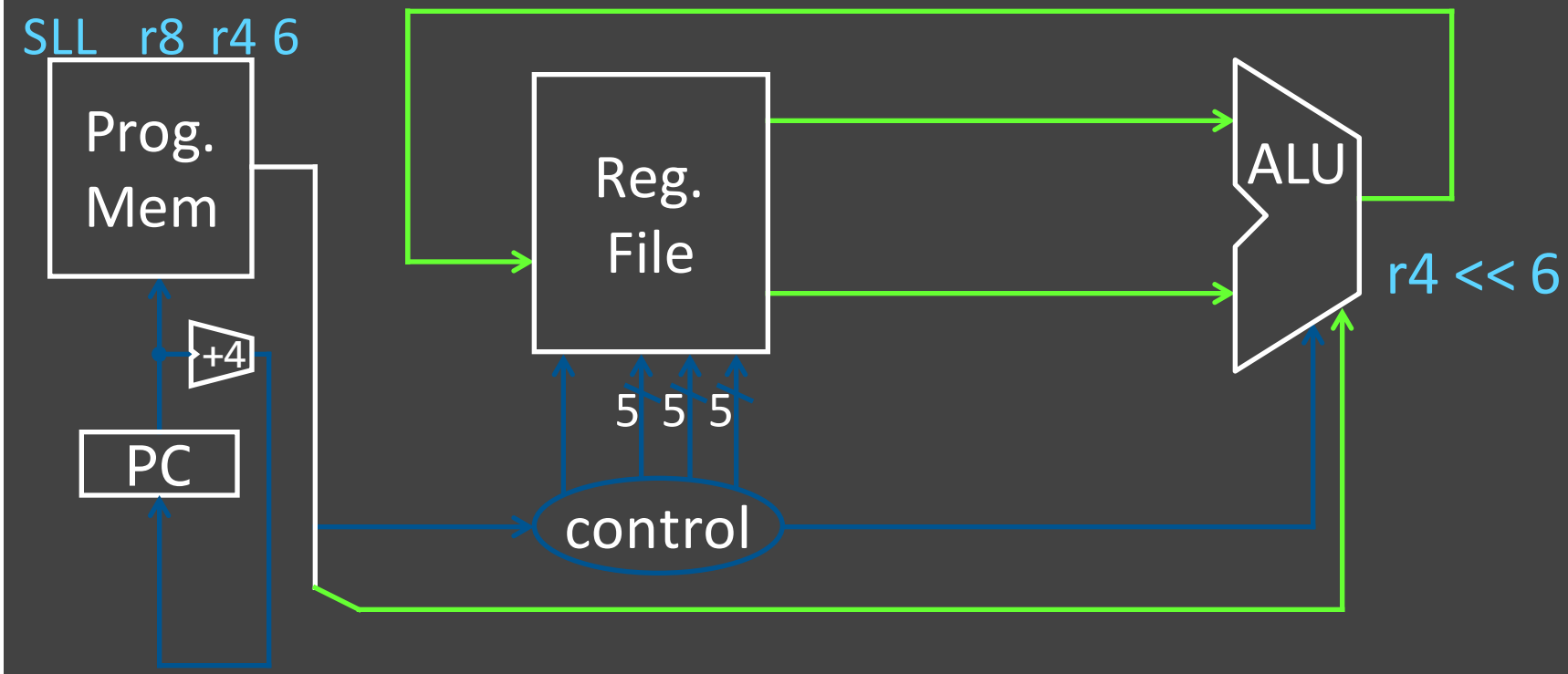
R-Type (2): Shift Instructions

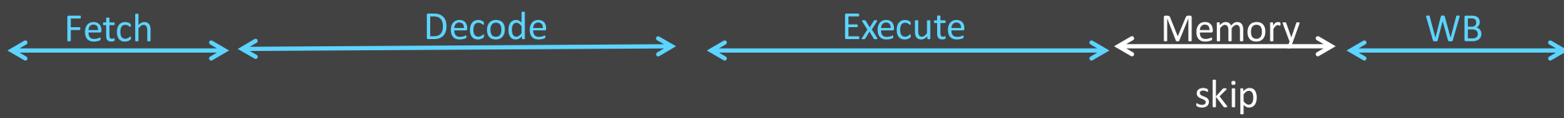
op	func	mnemonic	description
0x0	0x0	SLL rd, rt, shamt	R[rd] = R[rt] << shamt
0x0	0x2	SRL rd, rt, shamt	R[rd] = R[rt] >>> shamt (zero ext.)
0x0	0x3	SRA rd, rt, shamt	R[rd] = R[rt] >> shamt (sign ext.)

example: r8 = r4 * 64 # SLL r8, r4, 6

r8 = r4 << 6

Shift





Example: r8 = r4 * 64 # SLL r8, r4, 6

r8 = r4 << 6

I-Type (1): Arithmetic w/immediates

op	mnemonic	description
0x9	ADDIU rd, rs, imm	R[rd] = R[rs] + sign_extend(imm)
0xc	ANDI rd, rs, imm	R[rd] = R[rs] & zero_extend(imm)
0xd	ORI rd, rs, imm	R[rd] = R[rs] | zero_extend(imm)

example: r5 = r5 + 5 # ADDIU r5, r5, 5

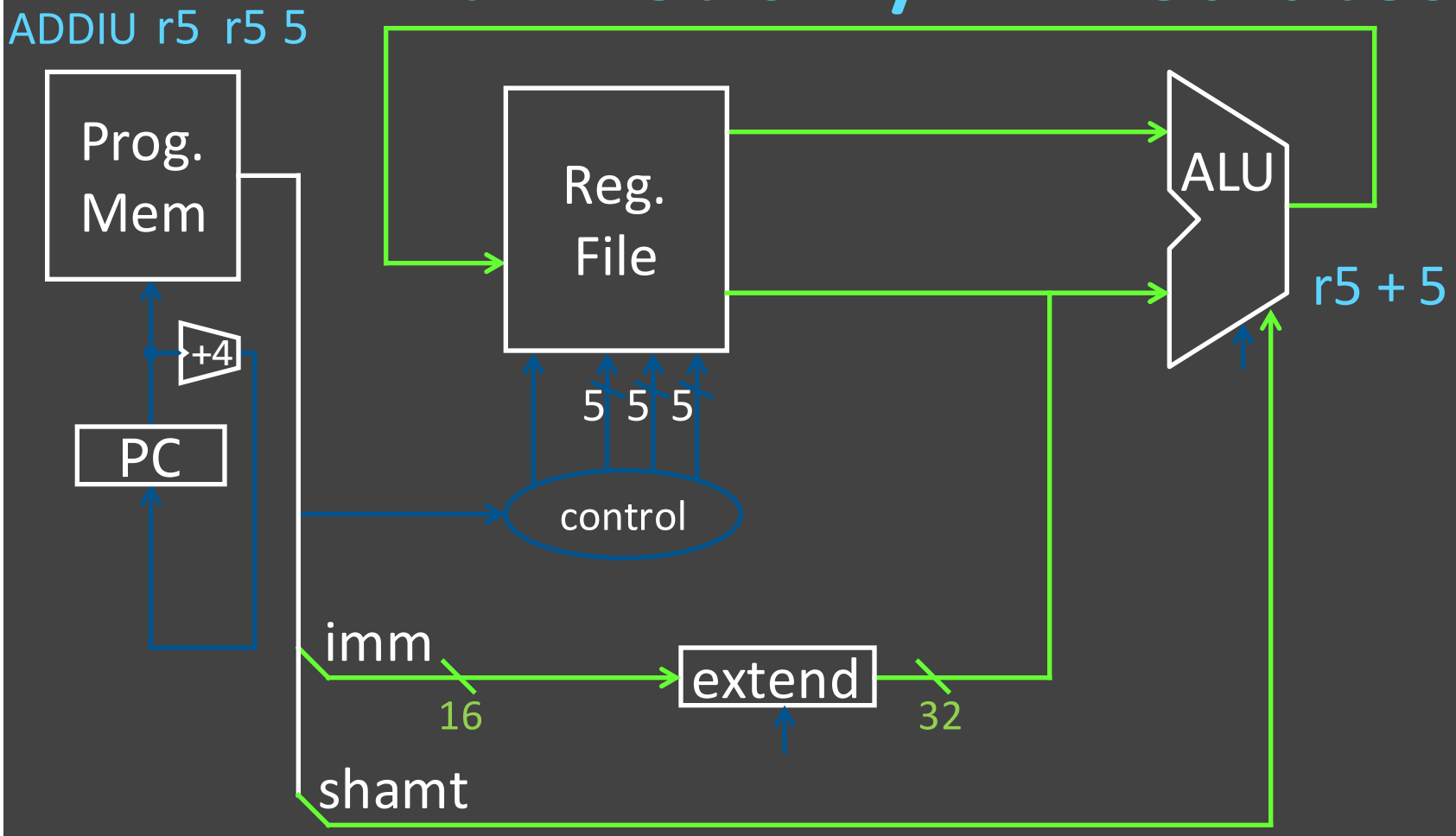
r5 += 5

What if immediate is negative?

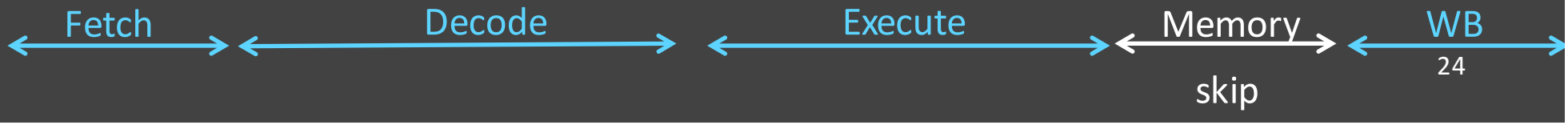
r5 += -1 r5 += 65535

Unsigned means no overflow detection.
The immediate *can* be negative!

Arithmetic w/immediates



Example: r5 = r5 + 5 # ADDIU r5, r5, 5



iClicker Question

Are you coming to the Homework 1 Review Session?

(A) Yes, I'm coming tonight (Tuesday).

(B) Yes, I'm coming tomorrow (Wednesday).

(C) Yes, but I don't know which night.

(D) Not sure yet.

(E) I won't be attending either.

I-Type (2): "Load" Upper Immediate

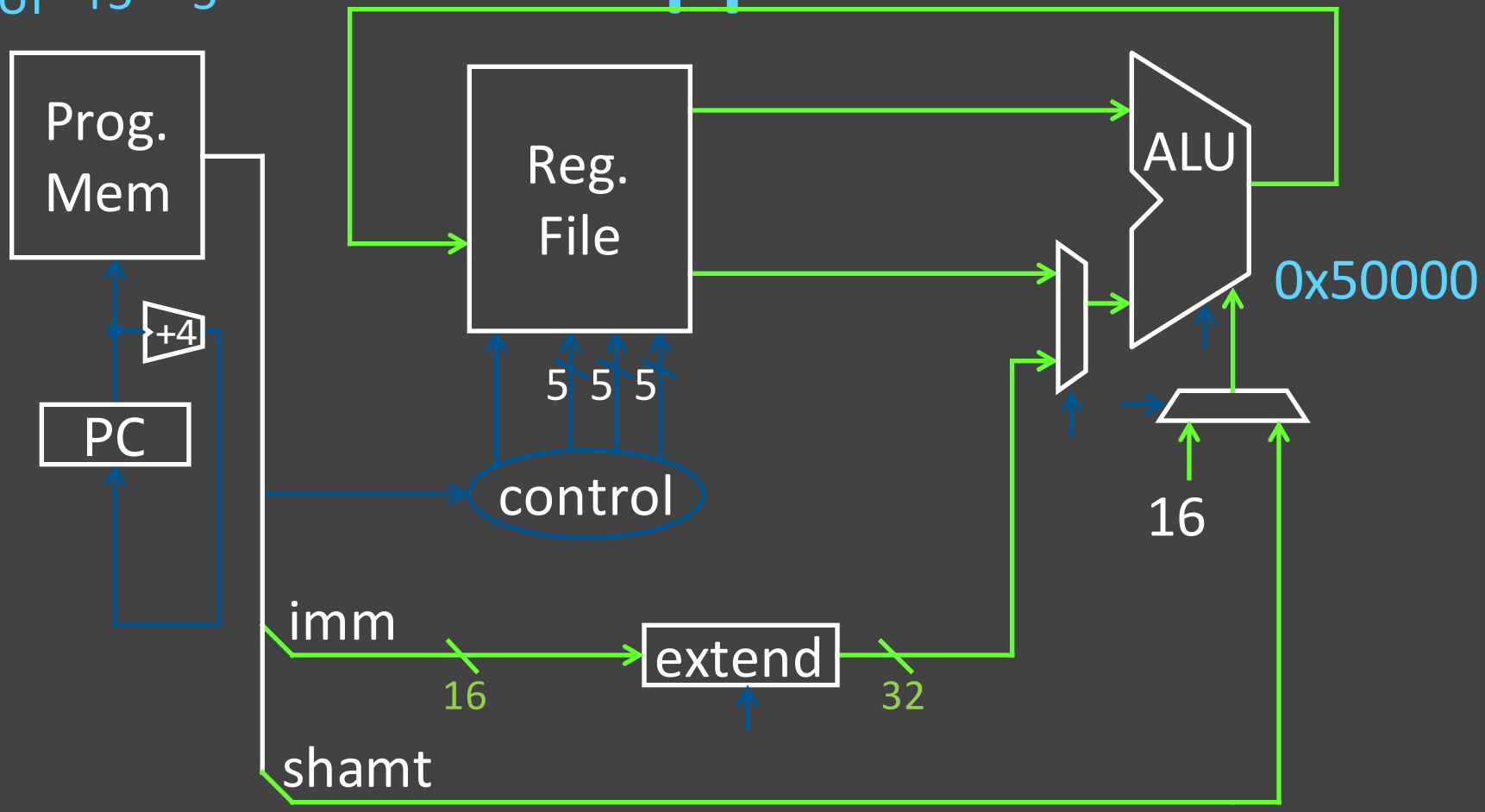
op	mnemonic	description
0xF	LUI rd, imm	R[rd] = imm << 16

example: r5 = 0x50000 # LUI r5, 5

Example:
```
LUI  r5,  0xdead
ORI  r5,  r5  0xbeef
```
What does r5 = ?

Load Upper Immediate

LUI r5 5



Example: r5 = 0x50000 # LUI r5, 5

Fetch | Decode | Execute | Memory | WB

skip

MIPS Instruction Types

Arithmetic/Logical

- R-type: result and two source registers, shift amount
- I-type: 16-bit immediate with sign/zero extension

Memory Access

- I-type
- load/store between registers and memory
- word, half-word and byte operations

Control flow

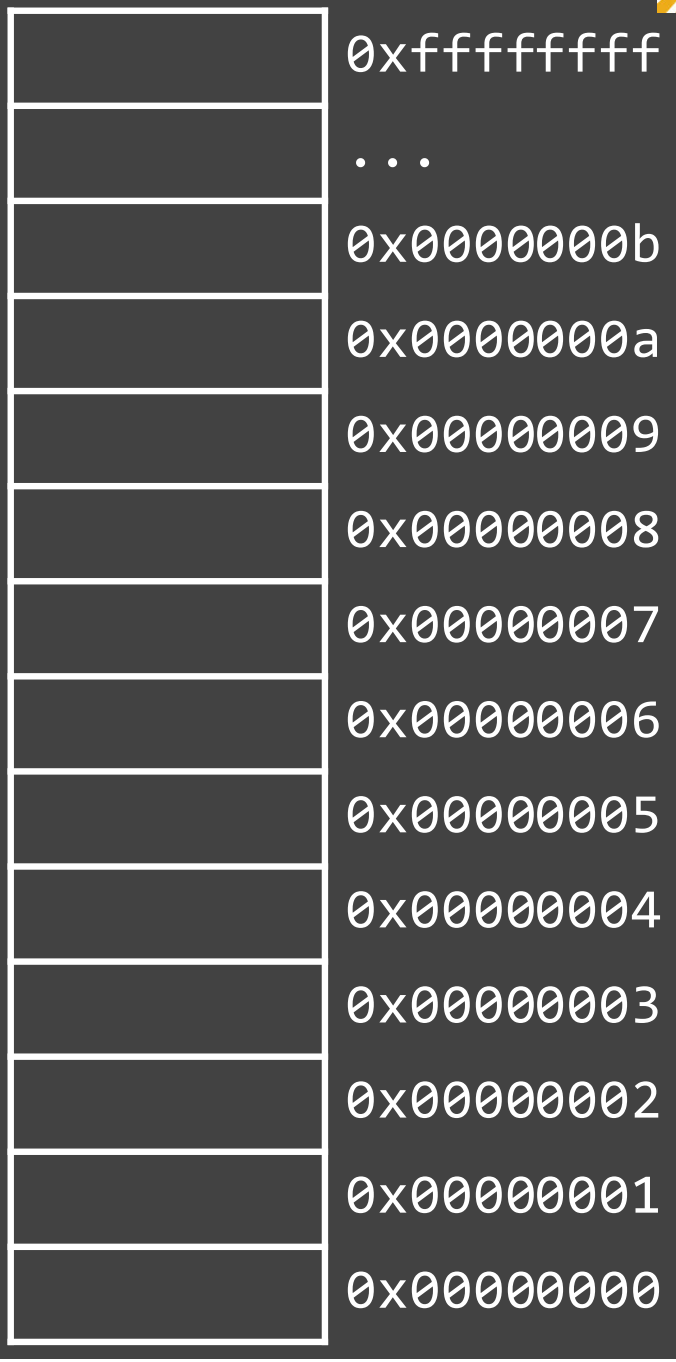
- J-type: fixed offset jumps, jump-and-link
- R-type: register absolute jumps
- I-type: conditional branches: pc-relative addresses

Memory Layout Options

```
# r5 contains 5 (0x00000005)

SB r5, 0(r0)
SB r5, 2(r0)
SW r5, 8(r0)
```

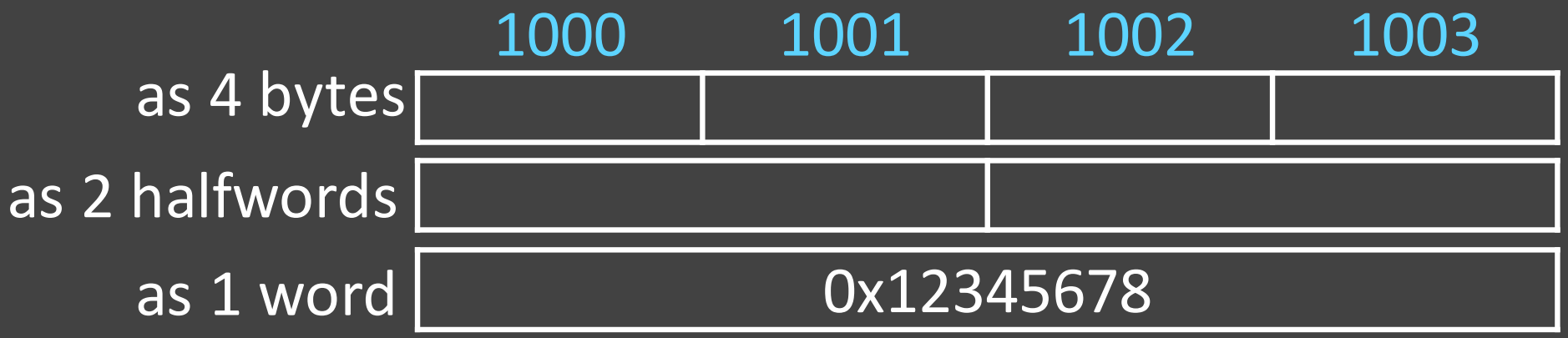
Two ways to store a word in memory.



Endianness

Endianness: Ordering of bytes within a memory word

Little Endian = least significant part first (MIPS, x86)



Big Endian = most significant part first (MIPS, networks)

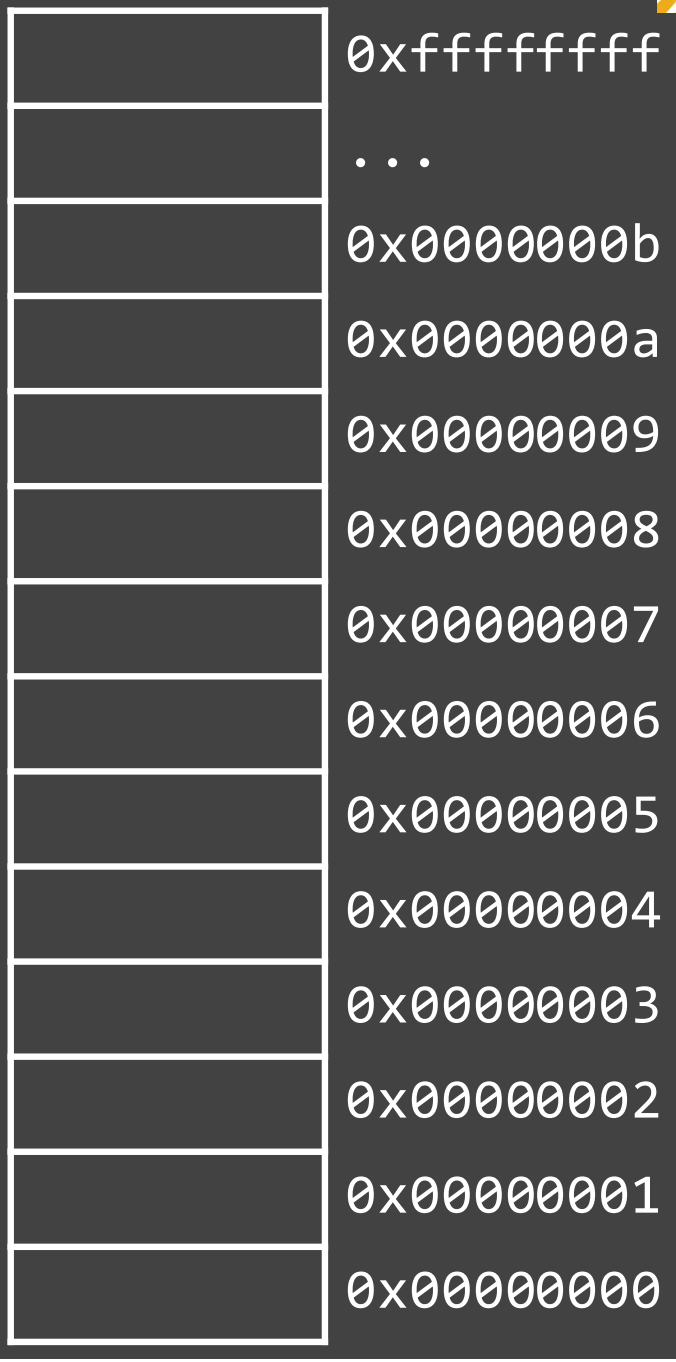
Big Endian Memory Layout

```
# r5 contains 5 (0x00000005)


SB r5, 2(r0)
LB r6, 2(r0)



SW r5, 8(r0)
LB r7, 8(r0)
LB r8, 11(r0)
```



I-Type (3): Memory Instructions

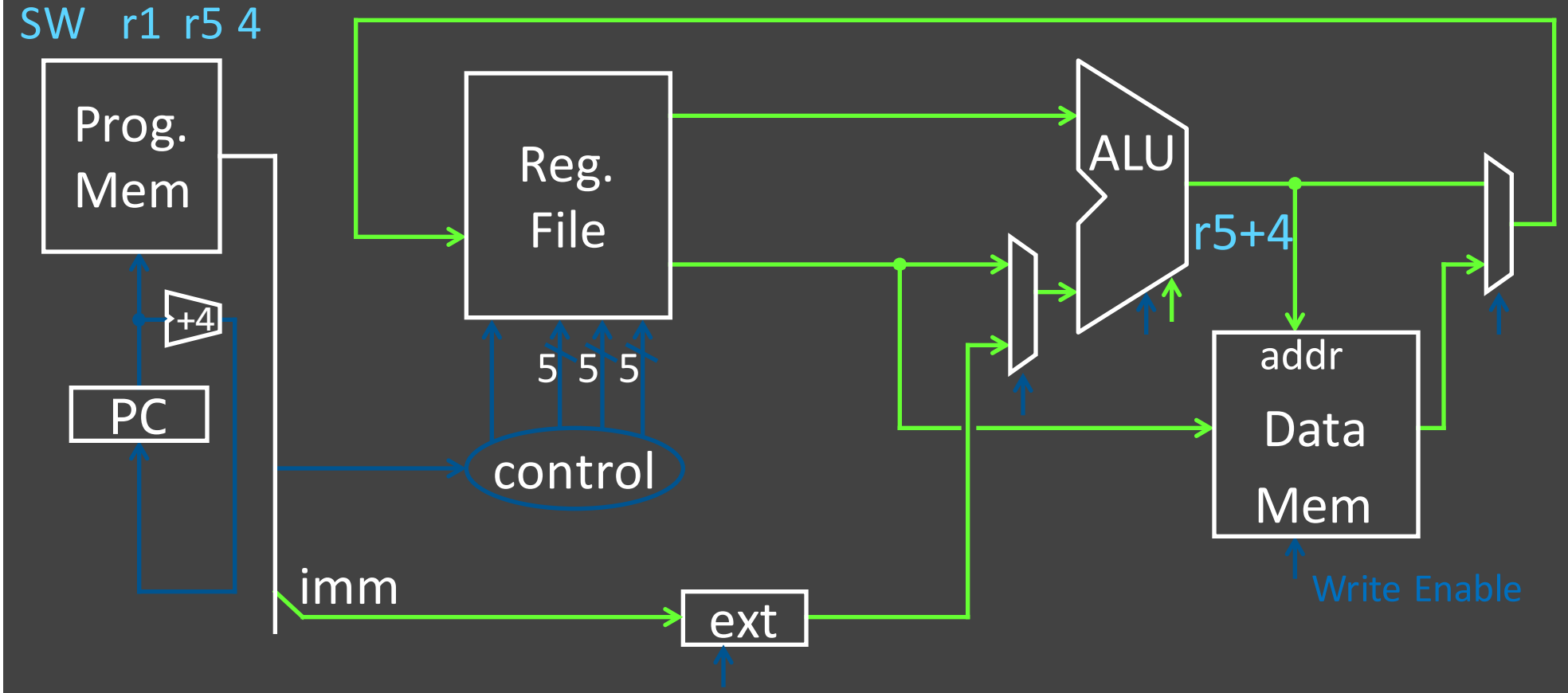
base + offset addressing

op	mnemonic	description
0x23	LW rd, offset(rs)	R[rd] = Mem[offset+R[rs]]
0x2b	SW rd, offset(rs)	Mem[offset+R[rs]] = R[rd]

signed offsets

Example: = Mem[4+r5] = r1 # SW r1, 4(r5)

Memory Operations



Example: = Mem[4+r5] = r1 # SW r1, 4(r5)

More Memory Instructions

op	mnemonic	description
0x20	LB rd, offset(rs)	R[rd] = sign_ext(Mem[offset+R[rs]])
0x24	LBU rd, offset(rs)	R[rd] = zero_ext(Mem[offset+R[rs]])
0x21	LH rd, offset(rs)	R[rd] = sign_ext(Mem[offset+R[rs]])
0x25	LHU rd, offset(rs)	R[rd] = zero_ext(Mem[offset+R[rs]])
0x23	LW rd, offset(rs)	R[rd] = Mem[offset+R[rs]]
0x28	SB rd, offset(rs)	Mem[offset+R[rs]] = R[rd]
0x29	SH rd, offset(rs)	Mem[offset+R[rs]] = R[rd]
0x2b	SW rd, offset(rs)	Mem[offset+R[rs]] = R[rd]

MIPS Instruction Types

Arithmetic/Logical

- R-type: result and two source registers, shift amount
- I-type: 16-bit immediate with sign/zero extension

Memory Access

- I-type
- load/store between registers and memory
- word, half-word and byte operations

Control flow

- J-type: fixed offset jumps, jump-and-link
- R-type: register absolute jumps
- I-type: conditional branches: pc-relative addresses

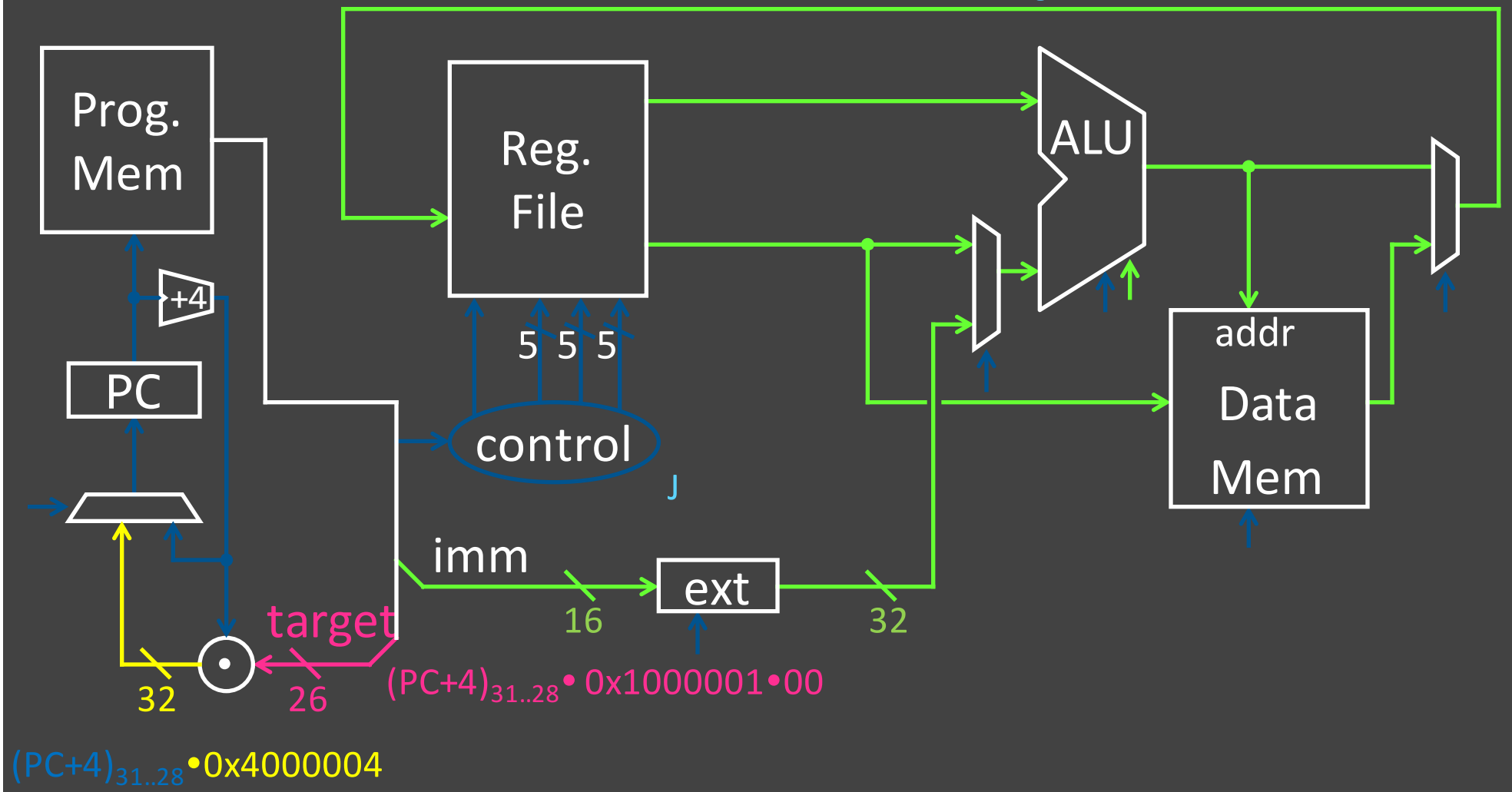
J-Type (1): Absolute Jump

op	Mnemonic	Description	"•" = *concatenate*
0x2	J target	PC = $(PC+4)_{31..28}$ • target • 00	

MIPS Quirk:

jump targets computed using *already incremented* PC

Absolute Jump



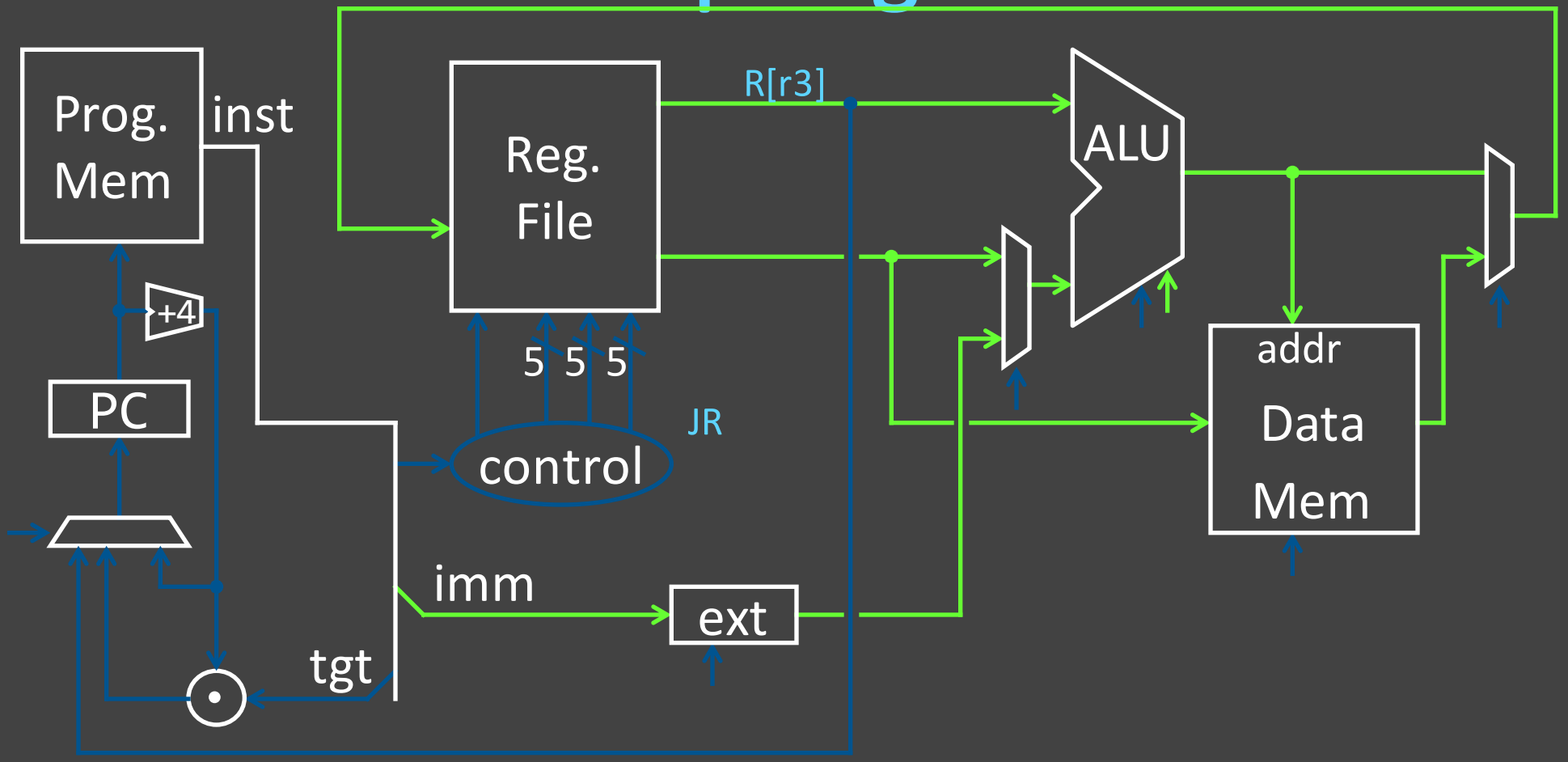
Example: PC = $(PC+4)_{31..28}$ • target • 00 # J 0x1000001

R-Type (3): Jump Register

op	func	mnemonic	description
0x0	0x08	JR rs	PC = R[rs]

Example: JR r3

Jump Register



ex: JR r3

op	func	mnemonic	description
0x0	0x08	JR rs	PC = R[rs]

Moving Beyond Jumps

Can use Jump or Jump Register instruction to jump to 0xabcd1234

What about a jump based on a condition?

```
# assume 0 <= r3 <= 1
if (r3 == 0) jump to 0xdecafe00
else jump to 0xabcd1234
```

I-Type (4): Branches

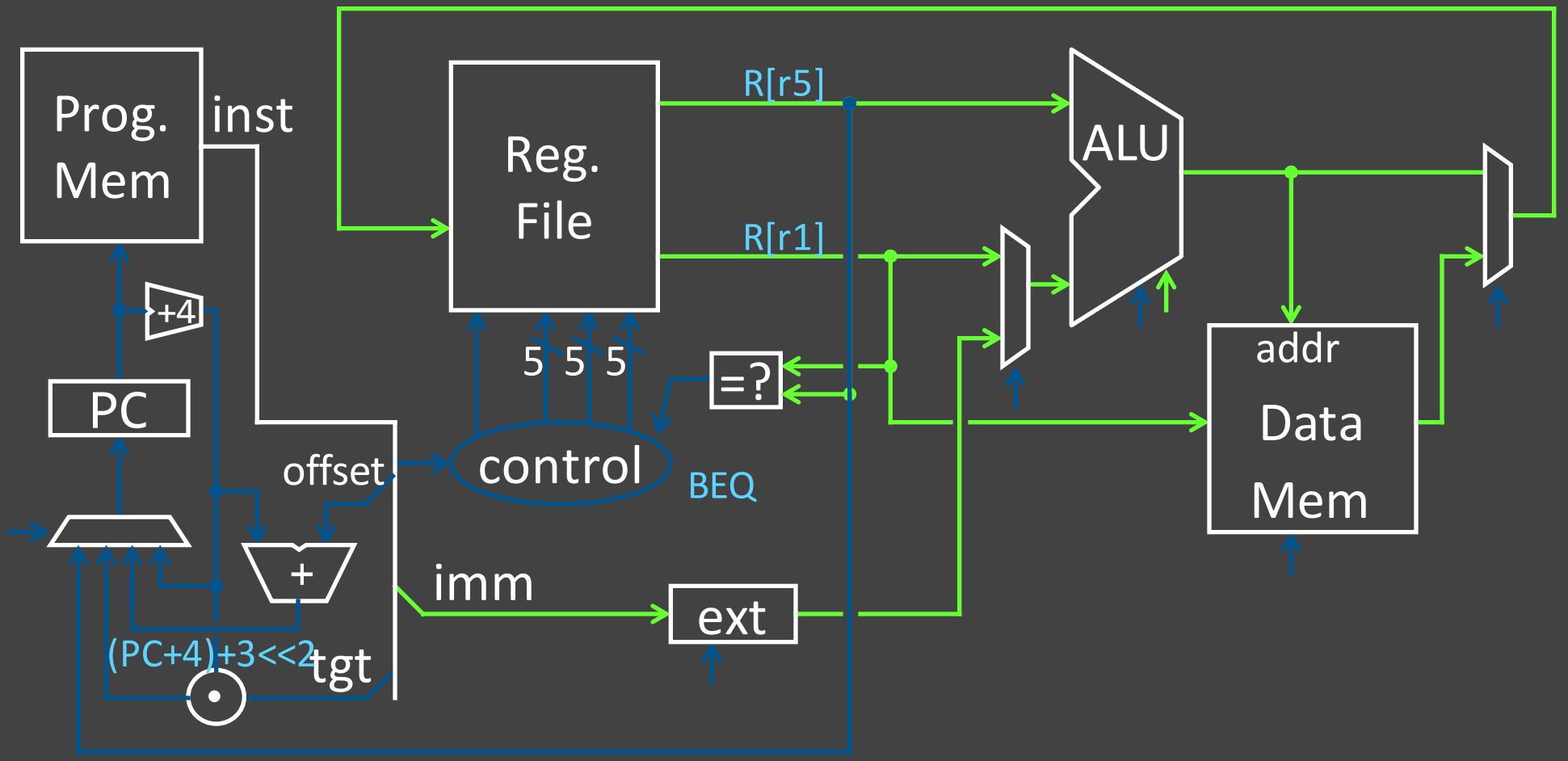
op	mnemonic	description
0x4	BEQ rs, rd, offset	if R[rs] == R[rd] then PC = PC+4 + (offset<<2)
0x5	BNE rs, rd, offset	if R[rs] != R[rd] then PC = PC+4 + (offset<<2)

Example: BEQ r5, r1, 3

```
if(R[r5]==R[r1])
      PC = PC+4 + 12   (i.e. 12 == 3<<2)
```

A word about all these +'s…

Control Flow: Branches



ex: BEQ r5, r1, 3

op	mnemonic	description
0x4	BEQ rs, rd, offset	if R[rs] == R[rd] then PC = PC+4 + (offset<<2)

I-Type (5): Conditional Jumps

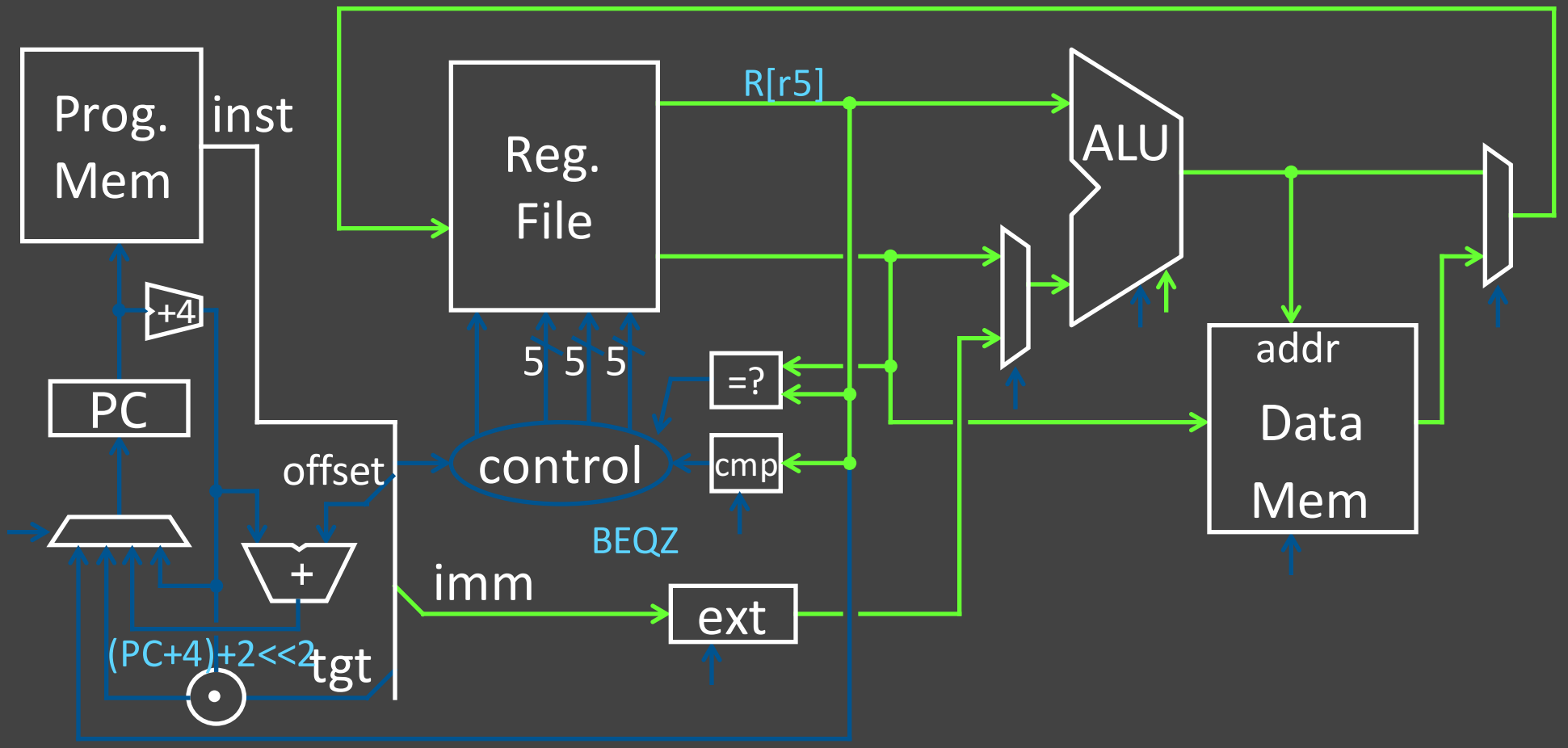
signed

op	subop	mnemonic	description
0x1	0x0	BLTZ rs, offset	if R[rs] < 0 then PC = PC+4+ (offset<<2)
0x1	0x1	BGEZ rs, offset	if R[rs] ≥ 0 then PC = PC+4+ (offset<<2)
0x6	0x0	BLEZ rs, offset	if R[rs] ≤ 0 then PC = PC+4+ (offset<<2)
0x7	0x0	BGTZ rs, offset	if R[rs] > 0 then PC = PC+4+ (offset<<2)

Example: BGEZ r5, 2

```
if(R[r5] ≥ 0)
    PC = PC+4 + 8   (i.e. 8 == 2<<2)
```

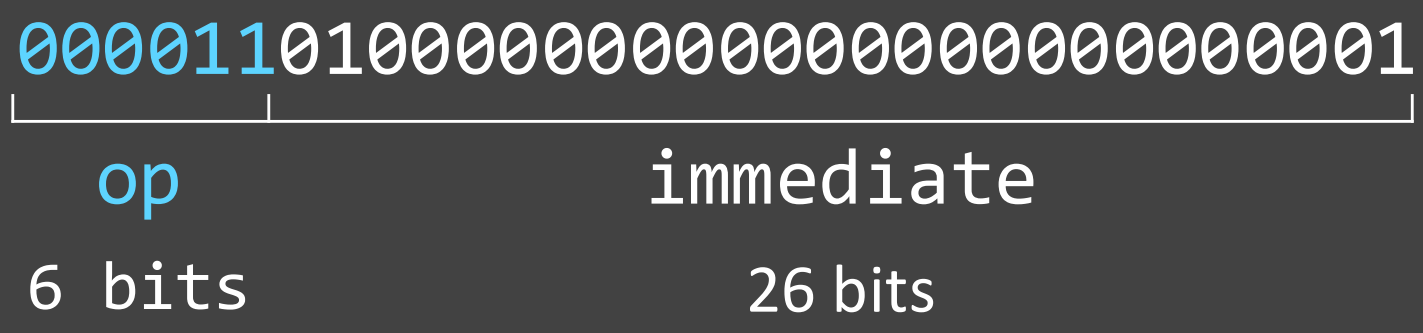
Control Flow: More Branches



ex: BGEZ r5, 2

op	subop	mnemonic	description
0x1	0x1	BGEZ rs, offset	if R[rs] ≥ 0 then PC = PC+4+ (offset<<2)

J-Type (2): Jump and Link



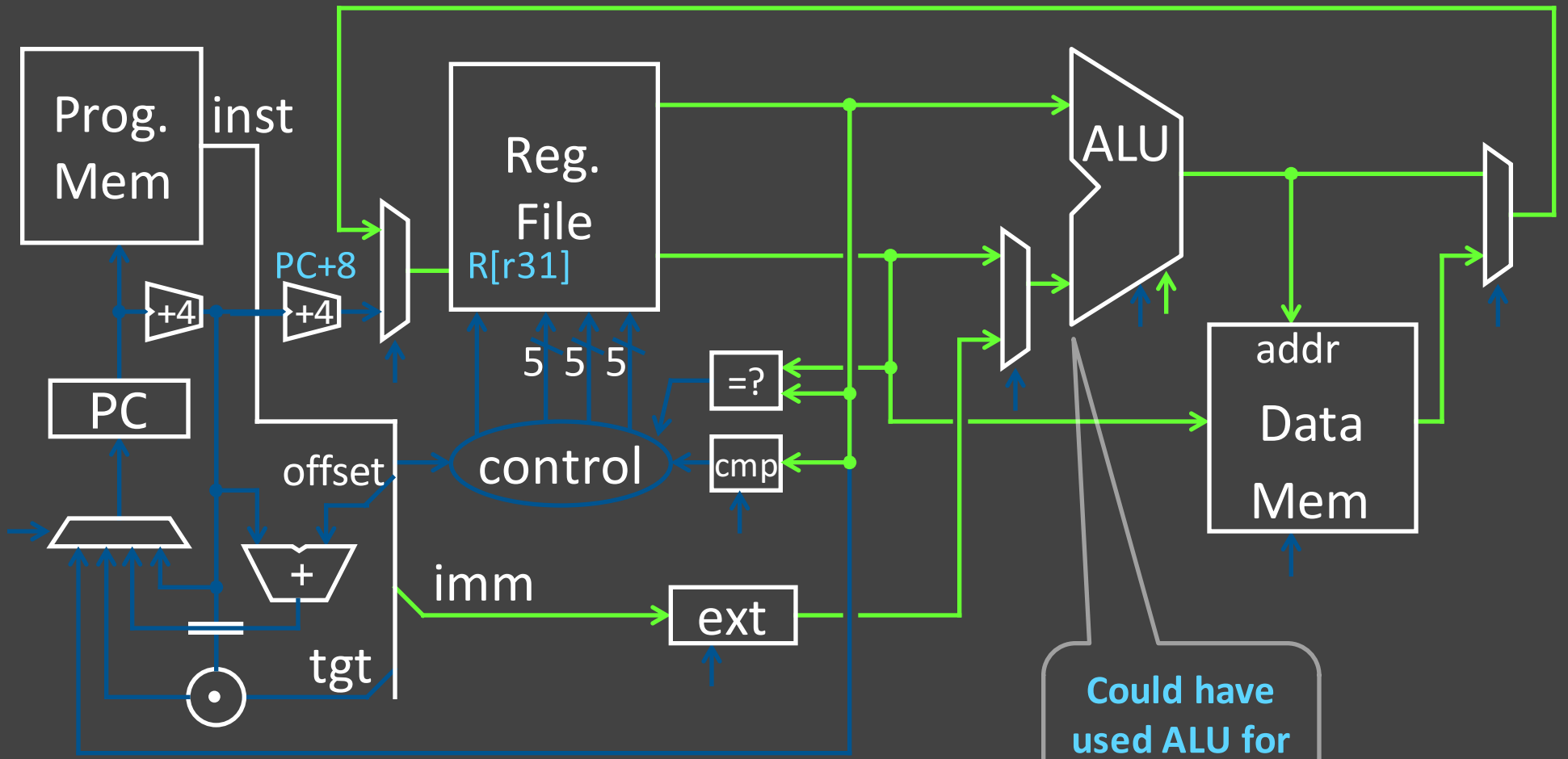
op	mnemonic	description
0x3	JAL target	r31 = PC+8 (+8 due to branch delay slot) PC = $(PC+4)_{31..28}$ • target • 00

Discuss later

Why?

Function/procedure calls

Jump and Link



ex: JAL 0x1000001 r31 = PC+8

$PC = (PC+4)_{31..28} \bullet 0x4000004$

op	mnemonic	description
0x3	JAL target	r31 = PC+8 (+8 due to branch delay slot) $PC = (PC+4)_{31..28} \bullet (target << 2)$

MIPS Instruction Types

✔ Arithmetic/Logical
- R-type: result and two source registers, shift amount
- I-type: 16-bit immediate with sign/zero extension

✔ Memory Access
- I-type
- load/store between registers and memory
- word, half-word and byte operations

✔ Control flow
- J-type: fixed offset jumps, jump-and-link
- R-type: register absolute jumps
- I-type: conditional branches: pc-relative addresses

Many other instructions possible:
- vector add/sub/mul/div, string operations
- manipulate coprocessor
- I/O

Summary

We have all that it takes to build a processor!

- Arithmetic Logic Unit (ALU)—Lab0 & 1, Lecture 2 & 3
- Register File—Lecture 4 and 5
- Memory—Lecture 5

MIPS 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