Basic Control Instructions

HW1 (written) grades published, HW2 due Wednesday night!

Outline

- Control instructions as software
- Adding control instructions to the data path
- Identifying and handling control hazards

Types of Branches

Software

```
int x = 7;
for (int i = 0; i < 4; i++) {
    x *= 2;
}
```

Perform each step of the loop assuming that the condition is true

Jump back up to the beginning of the loop declaration!

```
int x = 3;
do {
    x += 3;
} while (x < 30);
```

Perform each step of the loop assuming that the condition is true

Jump back up to the beginning of the loop declaration!

```
int x = 10;
if (x % 3 == 1) {
    return true;
} else {
    return false;
}
```

Depending on the condition, perform one side of the branch

1️⃣ Jump to else if false
2️⃣ Jump past else "if-block" if true
3️⃣ Return

Types of Branches

Software

- In general, if we want to manipulate where a program goes, our compiler will translate high-level code into *branch instructions* where the execution may be directed to some instruction that isn't the next to execute

- *Unconditional branches* imply that the next instruction to execute will always be at a well-defined next location

- *Conditional branches* describe instructions for which the control flow of the execution will depend on some data

Chat with your neighbor(s)!

Software

If we want to implement a branching instruction in assembly, what are the necessary fields to embed in the raw bytes for its execution?

Opcode to specify that we are implementing a branch

Destination address for the branch target

Register locations for input sources

Case Study: Branching in RISC-V

Software

Conditional Jumps

Unconditional Jumps

31–25	24–20	19–15	14–12	11–7	6–0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

1️⃣ bne r1, r2, 0xff00
2️⃣ blt r1, r2, 0xff00
"if-statements" and loops

1️⃣ jmp 0xff00
2️⃣ jalr
function calls and returns

Chat with your neighbor(s)!

bne, blt, jmp, jalr

Software

```
int x = 7;
for (int i = 0; i < 4; i++) {
    x *= 2;
}
```

```
1 ldi r1, 0
2 ldi r2, 1
3 ldi r3, 4
4 ldi r4, 7
5 ldi r5, 2
6 blt r1, r3, 10
7 add r1, r1, r2
8 mul r4, r4, r5
9 jmp 6
10 end
```

```
int x = 3;
do {
    x += 3;
} while (x < 30);
```

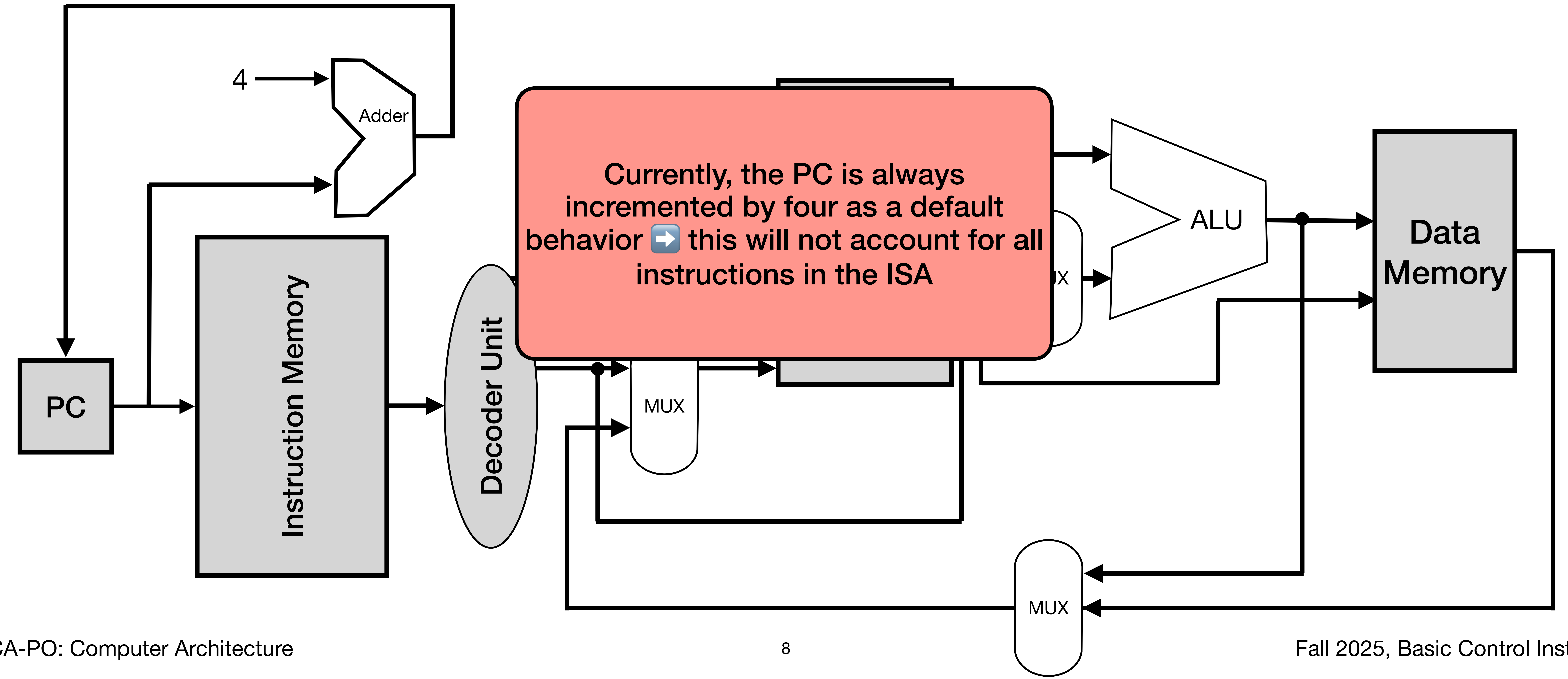
```
1 ldi r1, 3
2 ldi r2, 30
3 ldi r3, 3
4 add r3, r1, r3
5 blt r3, r2, 4
6 end
```

```
int x = 10;
if (x % 3 == 1) {
    return true;
} else {
    return false;
}
```

```
1 ldi r1, 10
2 ldi r2, 3
3 ldi r3, 1
4 mod r5, r1, r2
5 bne r5, r3, 8
6 ldi <return reg> 1
7 jalr
8 ldi <return reg> 0
9 jalr
```

Extending PC Update Logic

Hardware

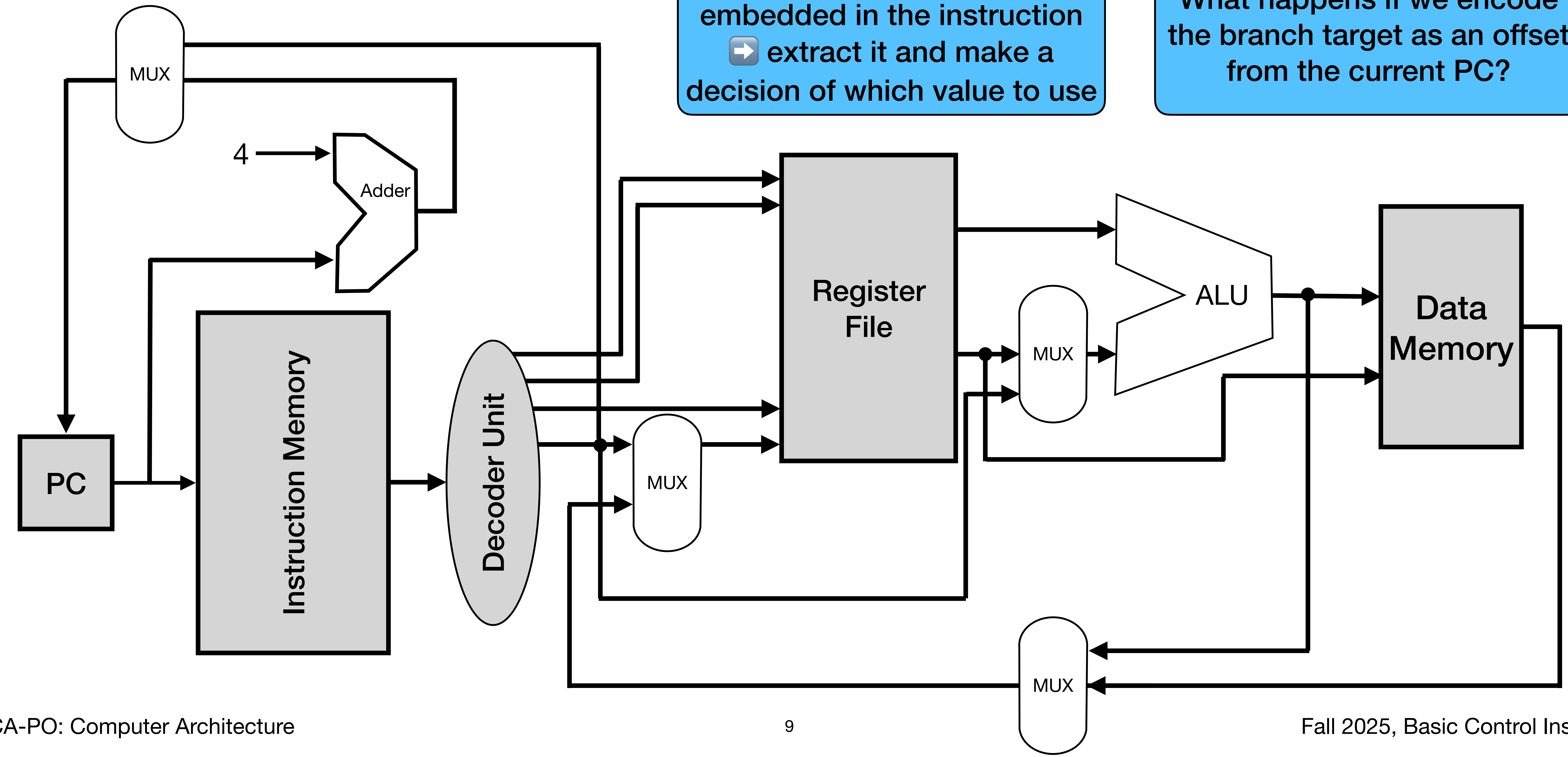


Updating the Data Path for Control

Hardware

The branch target is embedded in the instruction ➡️ extract it and make a decision of which value to use

What happens if we encode the branch target as an offset from the current PC?



Computing Branch Targets

- In most real instruction sets, branch targets are embedded as an offset from the current program counter
- This enables the program to jump *further* into the program space than embedding the raw instruction into the binary ➡️ also allows for dynamic address layout randomization by the operating system
- If this is the case, then we need to include additional components in the data path to perform this computation!

Chat with your neighbor(s)!

Suppose we are pipelining our data path to include branch instructions. In what stage does it make sense to satisfy the MUX where the PC gets updated?

If we want to update the PC, then we need to make sure that the condition has been evaluated ➡️ this means that we will need to perform a computation and interpret the output!

Control Hazards

- If it takes several cycles to know what the appropriate next program counter value should be, then it may be the case that our processor executes instructions that are incorrect relative to the expected program behavior
- Executing instructions on the incorrect side of a branch is called a *control hazard* as it will lead to incorrect instructions in the pipeline
- If our processor implements a hazard checking unit, then the unit must also check to see if incorrect instructions are in the pipeline due to control hazards and appropriately stall/bubble the stages

Takeaways

- By adding control instructions, our programs can become more robust but they also add complexity to the underlying hardware
- Updating the control flow requires new hardware logic to update the PC and pipelining logic must change accordingly
- By updating the pipeline, we introduce control hazards that must be mitigated