Basic Control Instructions

HW3 (Part 1) due
Wednesday;
Check In 5 on Friday

IBM SSEC (1948)

Image credit: https://en.wikipedia.org/wiki/IBM_SSEC

Had such small storage that the platform supported “self-modifying code” to reduce code size

Was developed well before caching, but causes nightmares for instruction cache/ data cache coherence!

Allowing for self-modifying code breaks all sorts of buffer overflow defenses…

Outline

- Wrapping up the memory system
- Control instructions as software
- Where our data path falls short with respect to control hazards
- Identifying and handling control hazards

Overview of the Memory System

- Applications are increasingly data intensive. Demands of the memory system are increasing at a faster rate than the technology can advance… we turn to strategic organizations of components and structures!
- To access memory, processors use a “port” interface that gives the illusion of an instruction memory and data memory component, but in practice memory accesses leverage the cache hierarchy
- Caches help give the illusion of a large and fast memory by using heuristics derived from typical application behaviors ➡️ from these behaviors, we construct caches with various associativities, replacement policies, and prefetching
- Seeing as caches may be private or shared, the hierarchy needs to implement mechanisms to keep shared data coherent across caches
- As a result of using caches, there exist timing side channels in which information about other programs may be leaked to malicious adversaries

Types of Branches


```
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

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

Types of Branches

- 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

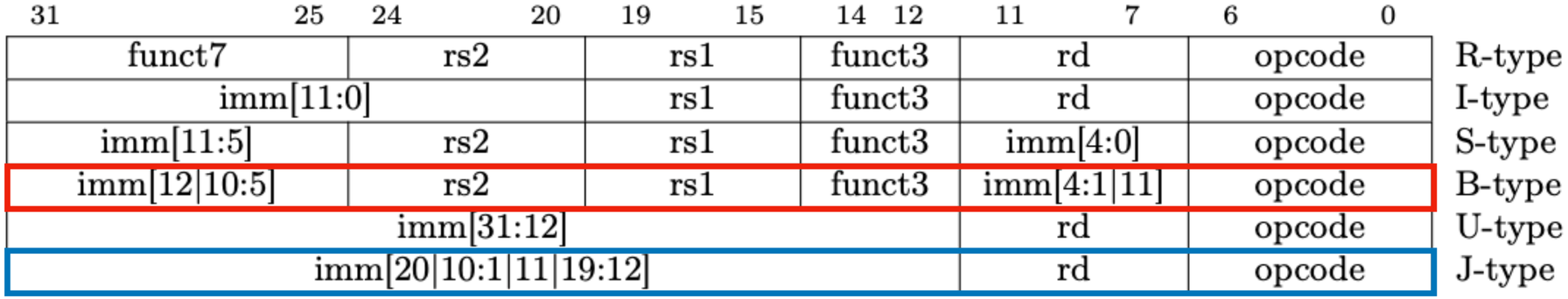
Destination address for the branch target

Register locations for input sources

Case Study: Branching in RISC-V

Conditional Jumps

Unconditional Jumps



1. `bne r1, r2, 0xff00`
2. `blt r1, r2, 0xff00`

"if-statements" and loops

1. `jmp 0xff00`
2. `jalr`

function calls and returns

What goes in the immediate field?

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, ???
7  add r1, r1, r2
8  mul r4, r4, r5
9  jmp ???
10 end
```

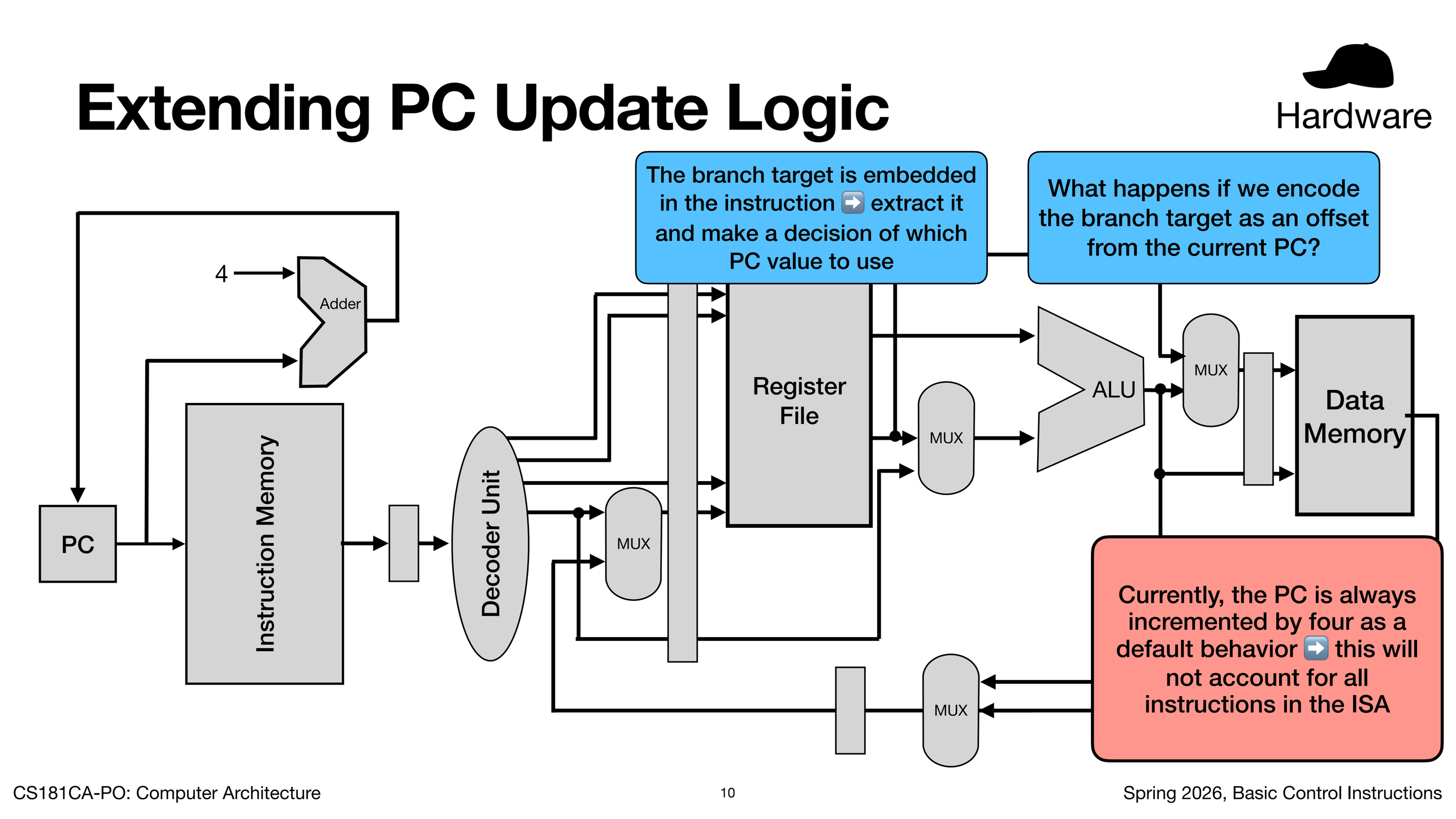
```
int x = 3;
do {
    x += 3;
} while (x < 30);
```

```
1  ldi r1, 3
2  ldi r2, 30
3  ldi r3, 3
4  add r3, r1, r3
5  blt r3, r2, ???
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, ???
6  ldi <return reg> 1
7  jalr
8  ldi <return reg> 0
9  jalr
```

Extending PC Update Logic



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. What combinational logic (e.g., component or gate) should we include to update the PC? In what stage does it make sense to satisfy the where the PC gets updated?

To make a decision on which output we should use, we need to have a multiplexor make the decision!

We cannot make the decision until target has been computed

In the case of conditional branches, we don't know the target unless the condition has been resolved

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