Advanced Pipelining: Branch Prediction, Exceptions, and Limits to Pipelining

Module 4

Module Syllabus

- Minimizing the impact of branches (control hazards)
- Handling exceptions
- The limits to pipelining

Control Hazards

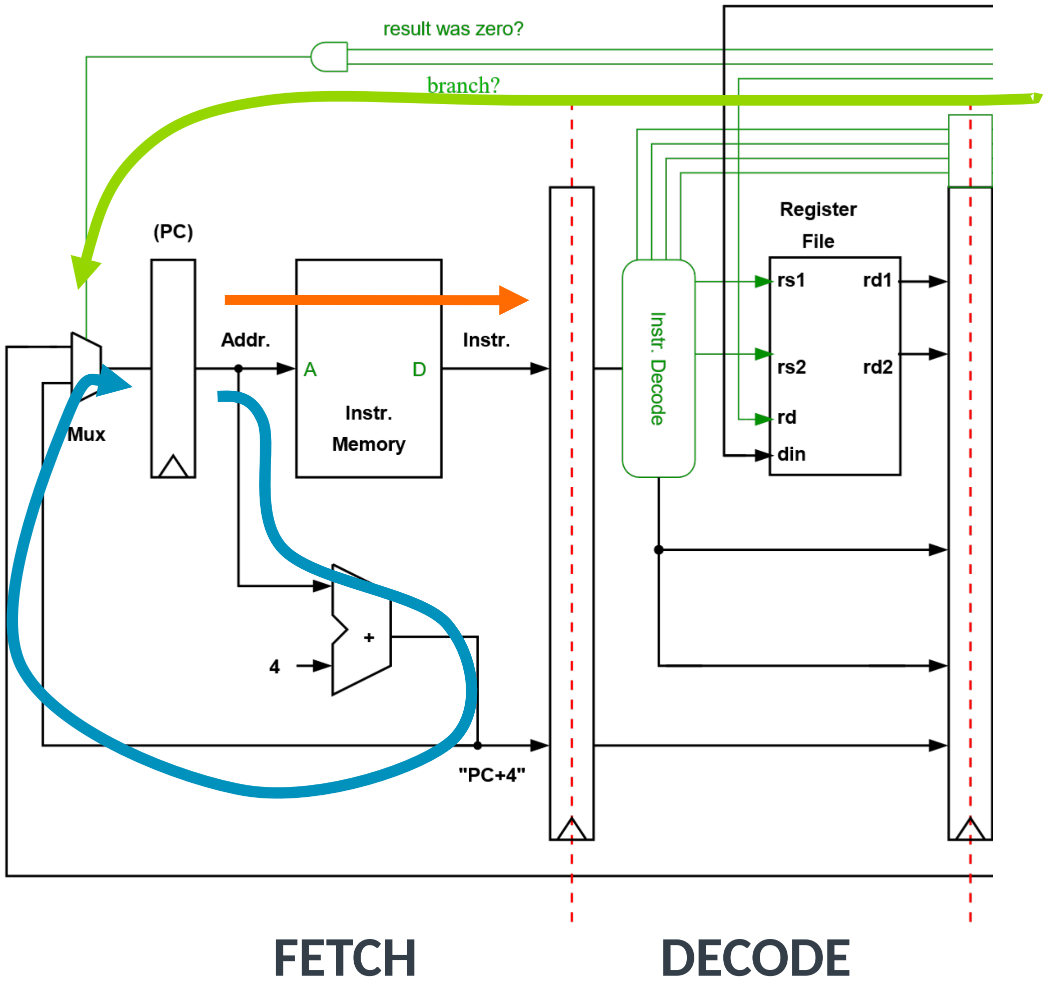
What happens immediately after we fetch a conditional branch instruction?

1. We must determine if the branch is taken or not.
2. If the branch is taken, we must compute and branch to the target address.

How Would We Like to Handle Branches?

In a pipelined processor, we would like to calculate the next value of the program counter (PC) in parallel with reading our instruction memory.

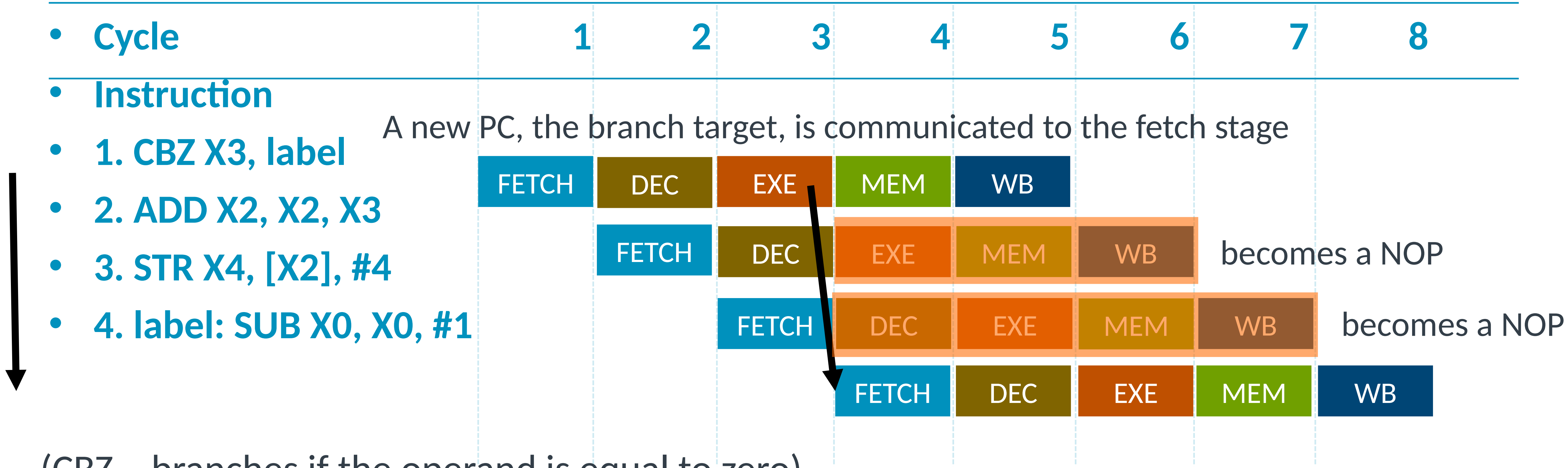
In practice, this is non-trivial to support as branches may not be decoded and evaluated until later in the pipeline.



How Would We Like to Handle Branches?

- Option 1: Assume a branch is not taken
- Option 2: Evaluate the branch earlier in the pipeline
- Option 3: Delayed branch
- Option 4: Branch prediction

Option 1: Assume the Branch Is Not Taken



Cycle	1	2	3	4	5	6	7	8
Instruction								
1. CBZ X3, label	FETCH	DEC	EXE	MEM	WB			
2. ADD X2, X2, X3		FETCH	DEC	EXE	MEM	WB		
3. STR X4, [X2], #4			FETCH	DEC	EXE	MEM	WB	
4. label: SUB X0, X0, #1				FETCH	DEC	EXE	MEM	WB

A new PC, the branch target, is communicated to the fetch stage

becomes a NOP

becomes a NOP

(CBZ – branches if the operand is equal to zero)

If the branch is evaluated in the execute stage, and it is taken, we must convert the two instructions that follow it into NOPs (we waste two cycles).

Option 1: Assume a Branch Is Not Taken

If we evaluate the branch in the execute stage, we would lose two cycles every time we encountered a taken branch.

If we assume 20% of instructions are branches, and 60% are taken, and an otherwise perfect CPI of 1:

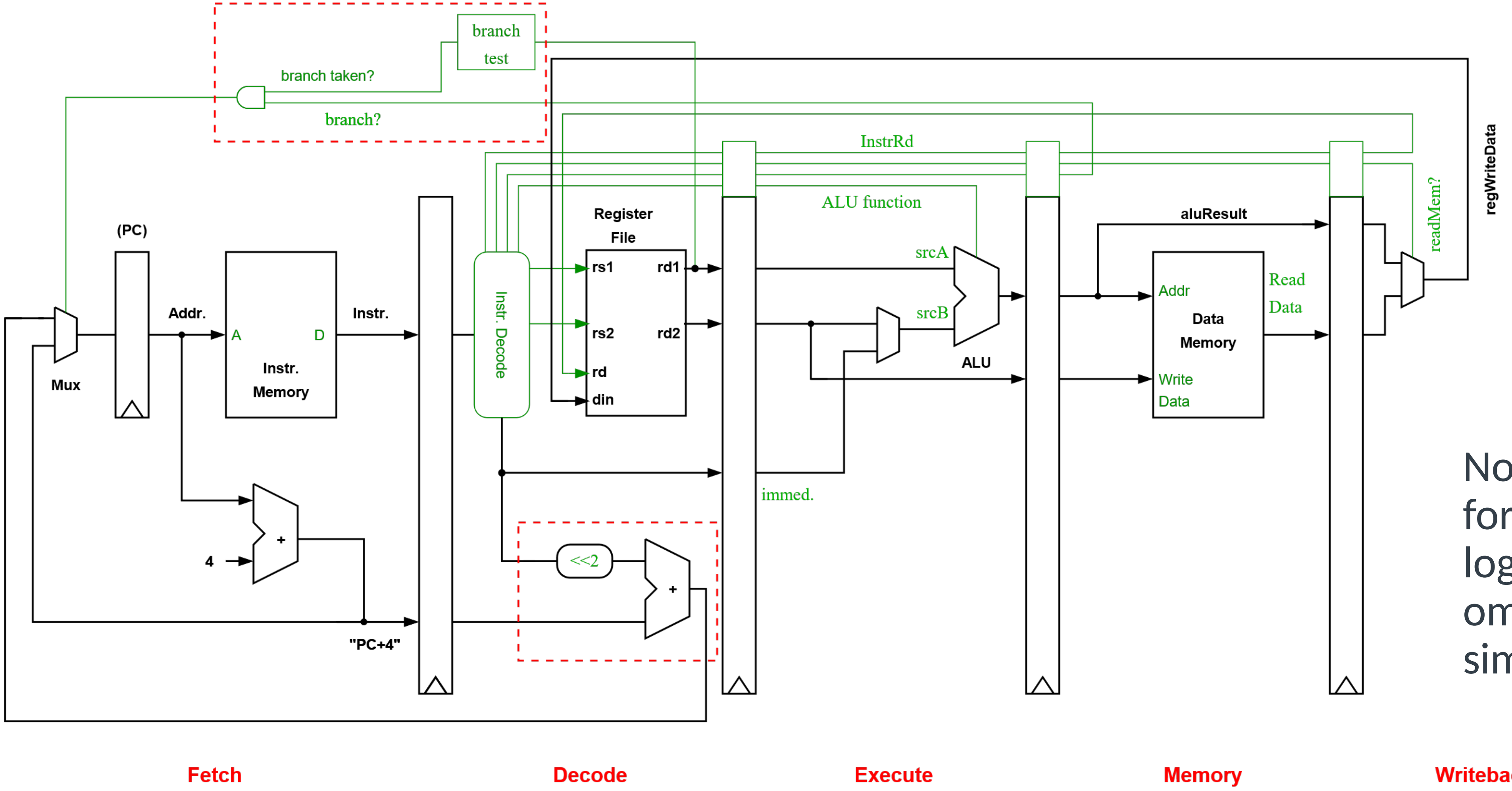
CPI contribution from taken branches = 0.2*2 * 0.6 = **0.24**

Our new CPI = 1.0 + branch stalls = 1.24

Option 2: Evaluate the Branch Earlier in the Pipeline

- Move the branch test and branch target address calculation to the decode stage.
- This would reduce the branch penalty to a single cycle in the case of a taken branch.
  - i.e., in the case of a taken branch, we would need to discard the instruction after the branch.

Option 2: Evaluate the Branch Earlier in the Pipeline



Note: Data forwarding logic omitted for simplicity

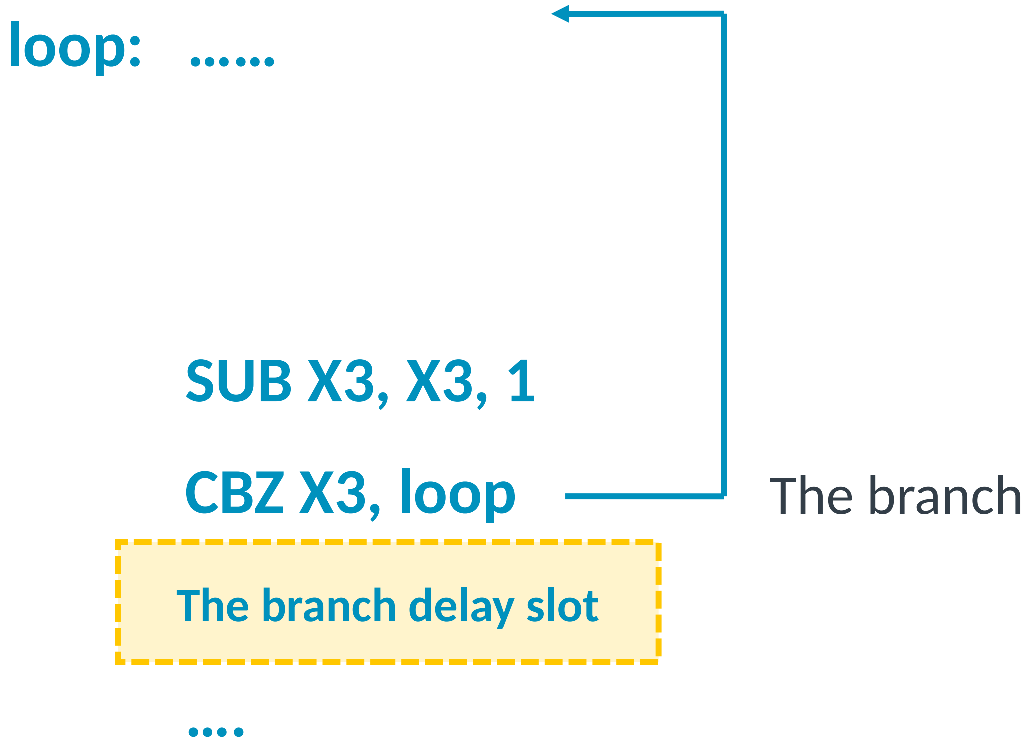
Option 2: Evaluate the Branch Earlier in the Pipeline

For this technique to work:

- The branch condition must be simple to evaluate.
    - Test for zero is simple .
    - Tests requiring an ALU operation are probably too complex.
    - We don’t want to the branch test to extend the clock cycle time.
- We must take care of potential data hazards.
    - If the instruction immediately before the branch writes to the register than the branch tests, we must stall for one cycle (i.e., until this instruction generates its result).
    - We will also need forwarding paths from the EXE and MEM pipeline stages to the decode stage.

Option 3: A Delayed Branch

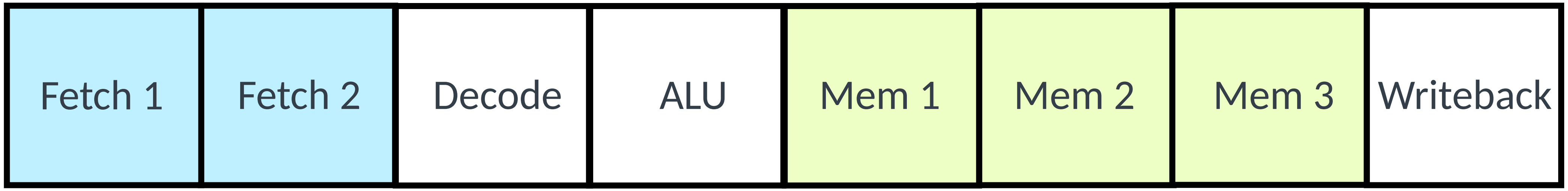
- We could decide to always execute the instruction after the branch, regardless of whether the branch is taken or not.
- This instruction after the branch is now called the **"branch delay slot."**



Option 3: A Delayed Branch

- A compiler can usually fill a single branch delay slot around 60-70% of the time.
- If we had a longer pipeline, we could introduce multiple branch delay slots, but they are typically hard to fill.
- Question: where can we find an instruction to move to the branch delay slot?
- If we can’t find an instruction to fill the slot, we have to fill it with a NOP.

Example



Fetch 1	Fetch 2	Decode	ALU	Mem 1	Mem 2	Mem 3	Writeback

This pipeline has 8 pipeline stages.

- The basic branch delay is 3 cycles.
- The branch condition and target address are evaluated in the ALU stage.
- All branches are predicted as not taken.

Example

- Impact on CPI:
  - Let’s make some reasonable assumptions:
  - 15% of instructions are unconditional or taken branches; the penalty here is 3 cycles.
  - 5% of instructions are untaken conditional branches; no penalty here, as we are predicting these not-taken.
  - **CPI contribution from branch stalls** = 0.15 * 3 + 0.05 * 0 = **0.45**

Option 4: Branch Prediction

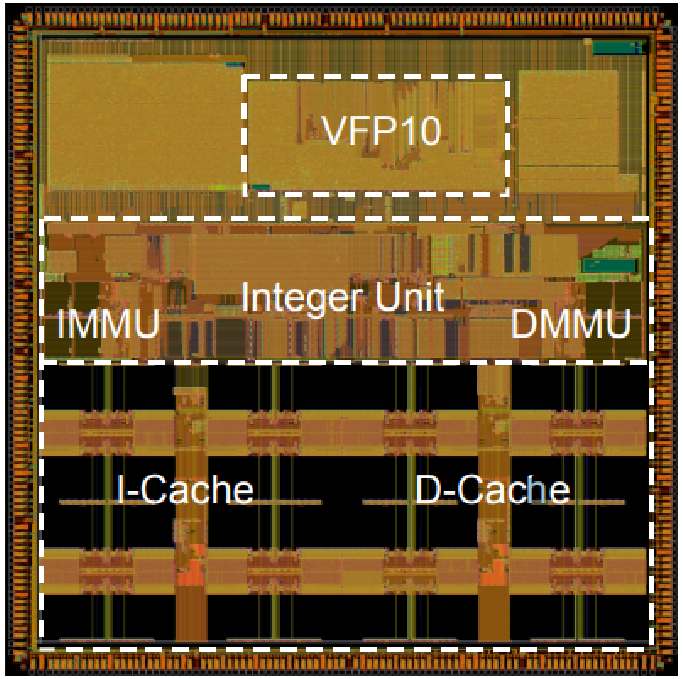
- For high-performance processors, with deep pipelines, the techniques described so far are inadequate.
- E.g., **Arm Cortex-A15**
  - 15-stage pipeline
  - Branches are evaluated late in the pipeline.
  - Branch misprediction penalty ~14 cycles
  - It also fetches 4 instructions per cycle and decodes 3 at a time (i.e., there are multiple instructions in a single pipeline stage).
  - We may discard>40 instructions if we must flush the pipeline due to a mispredicted branch.

Static Branch Prediction (Mostly Historic)

- Static prediction methods exploit the observation that a given branch instruction is likely to be highly biased in one direction.
- Schemes are often based on whether the branch is branching forward or backward in the code or alternatively depend on the op-code of the branch instruction.
- **Displacement-based prediction**
  - We can exploit a simple observation that backward branches are usually loop branches and are likely to be taken.
  - Now, if the branch's target address < PC, we predict the branch taken.
  - The accuracy of this sort of scheme is around 65% (or 80-95% with aid of profiling).

Example: Arm10 Processor

- The integer pipeline had 6 stages.
- The Arm10 employed a static displacement-based branch prediction scheme.
- The Arm10's Branch Prediction Unit also worked ahead of the fetch stage (prefetching instructions) in order to reduce the costs of branches.



Arm10 (Circa 2000)

Transistor count: ~250,000

Clock frequency: 400 MHz

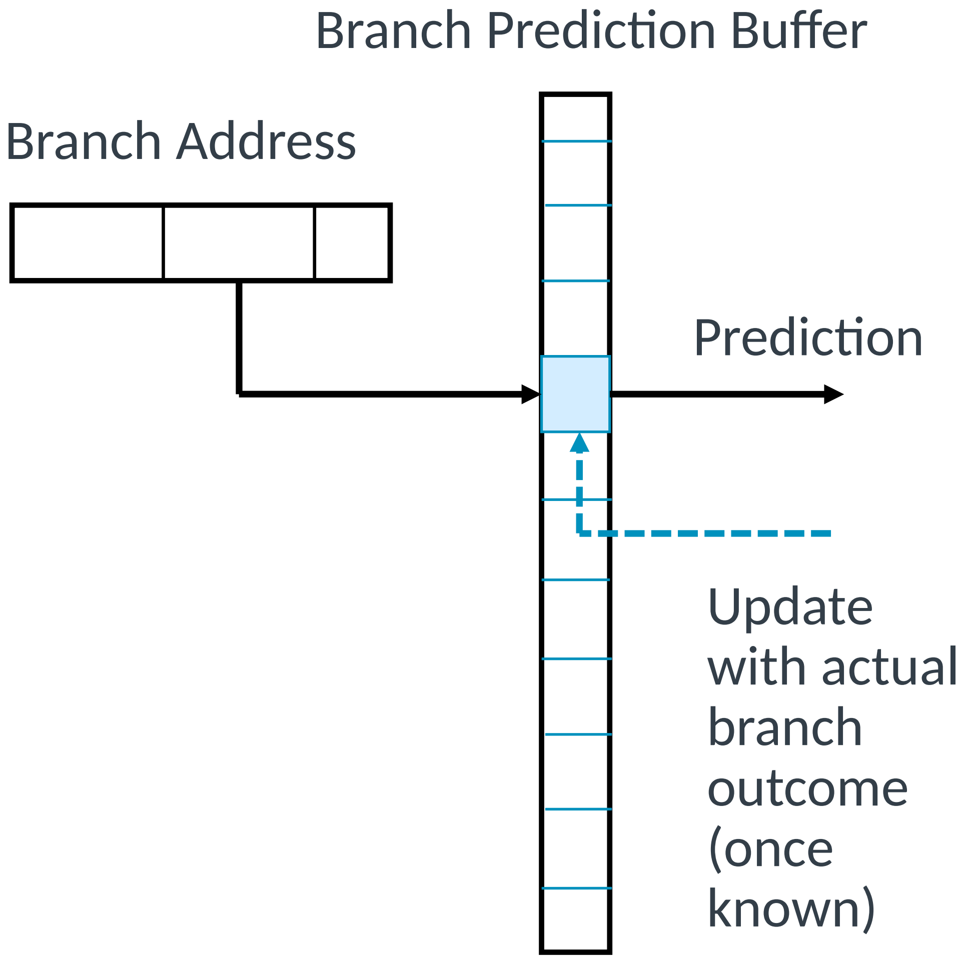
Technology: 0.18-micron

Applications: modems, cellular phones, automotive, etc.

A Simple One-level Dynamic Branch Predictor

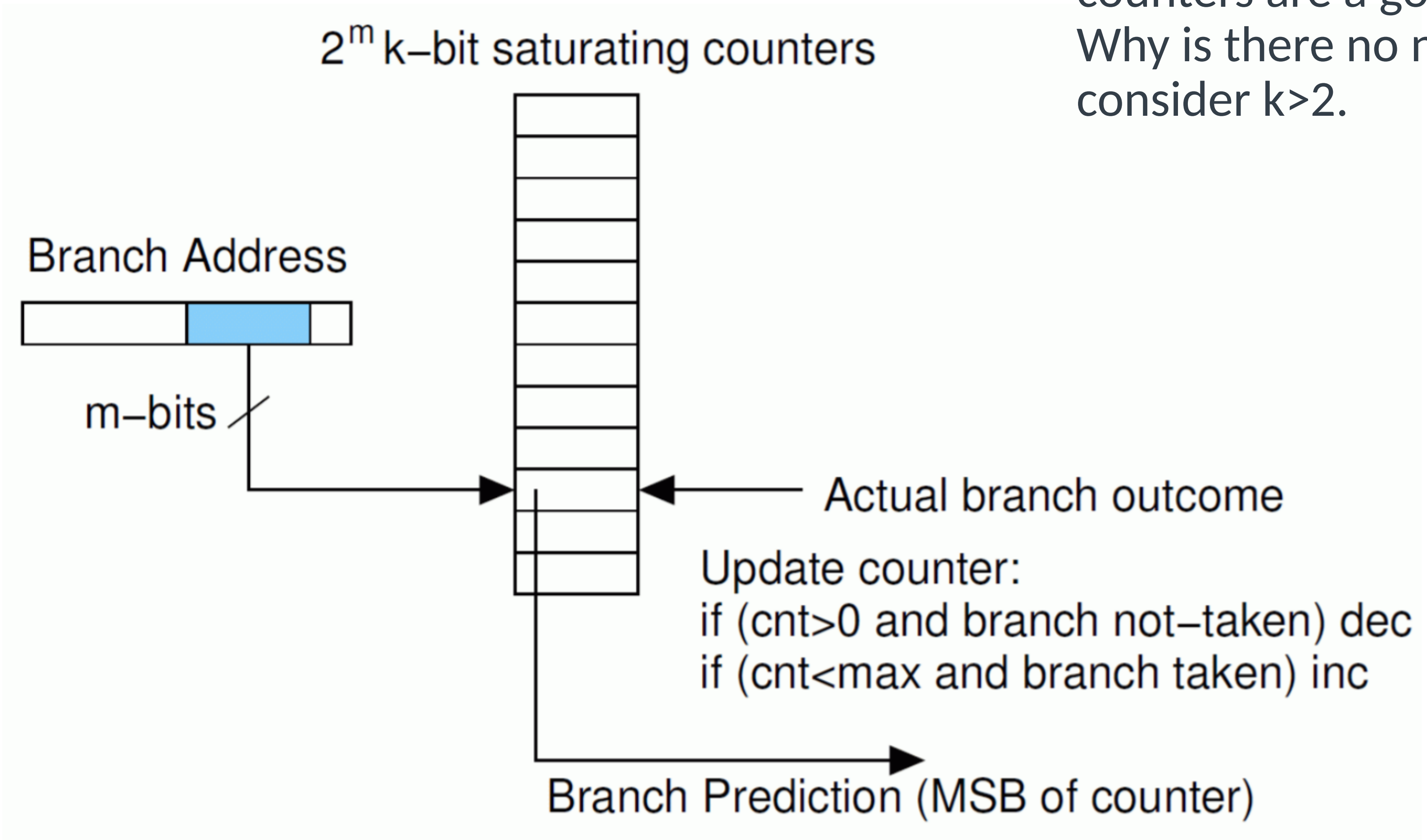
A simple dynamic branch prediction algorithm is to **predict a branch taken only if it was the latest time we executed it**. We can use a simple table of 1-bit entries to store our predictions

The drawback of these 1-bit table entries is that a single event can flip the prediction. We probably want some hysteresis, e.g., in the case of loops to avoid two mis-predictions per loop (upon entry and exit).



One-level Branch Predictors

In practice, 2-bit saturating counters are a good choice. Why is there no need to consider k>2.


2^m k–bit saturating counters
Branch Address
m–bits
Actual branch outcome
Branch Prediction (MSB of counter)

Update counter:
if (cnt>0 and branch not–taken) dec
if (cnt<max and branch taken) inc

Correlating Predictors

- How can we improve on simple bimodal predictors?
- We can take advantage of the fact that the outcome of many branches is correlated either with the past outcomes of the same branch (the local history) or with other recent branches (the global history).
- Now instead of simply hashing the PC to select a counter, we can include local or global branch history to improve our prediction.

Correlating Predictors (Local History)

If a particular branch’s outcome has a repetitive pattern, its local history can be used to improve prediction accuracy.

E.g., a particular branch instruction may be predictable if we look at its local history:

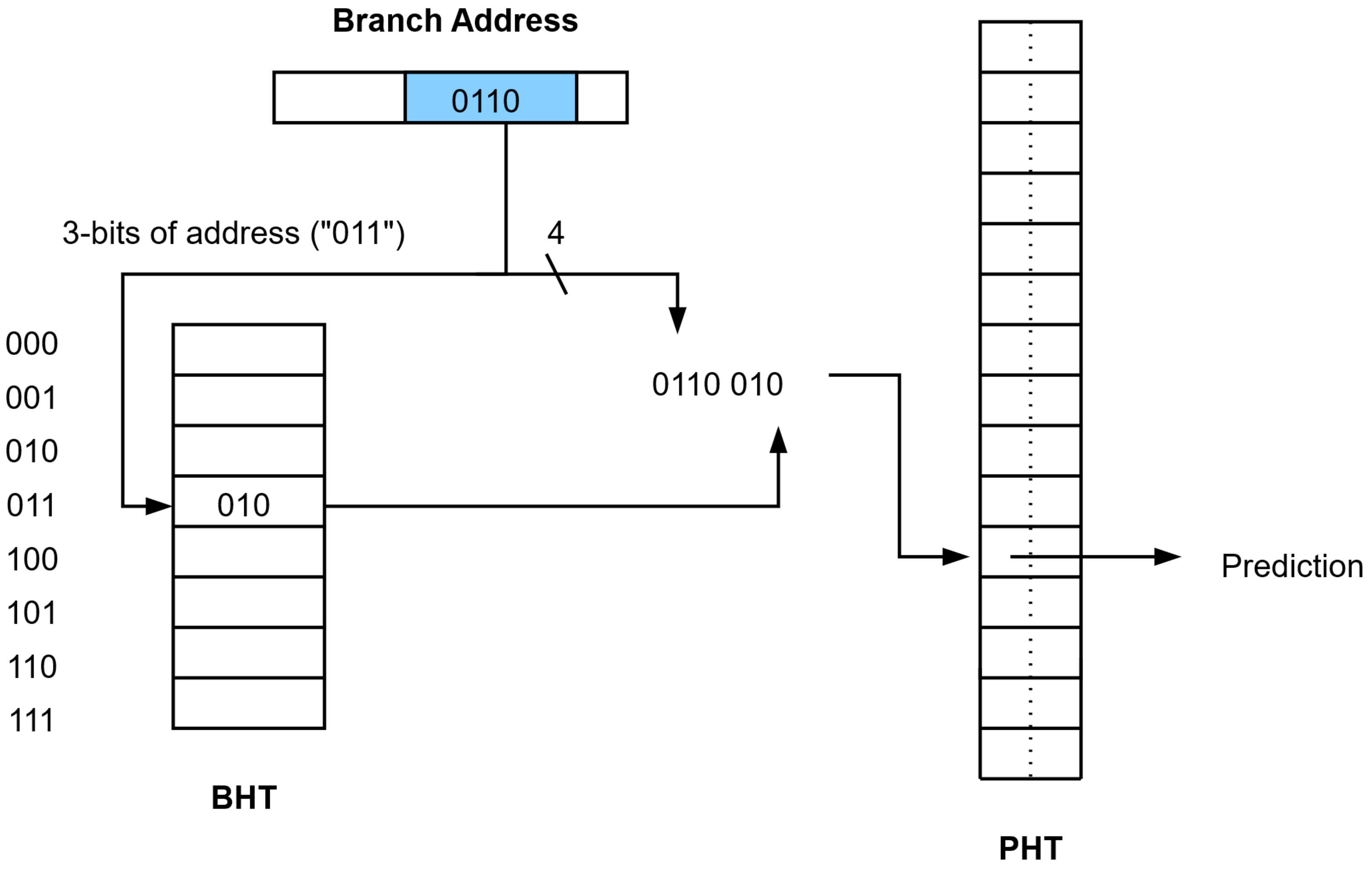
….010110101101011010110101101011

History	Next
0101	1
1011	0
0110	1
…	

Local-history Two-level Predictor

We now have two memories or tables:

- **The Branch History Table (BHT)**
- **The Pattern History Table (PHT)**

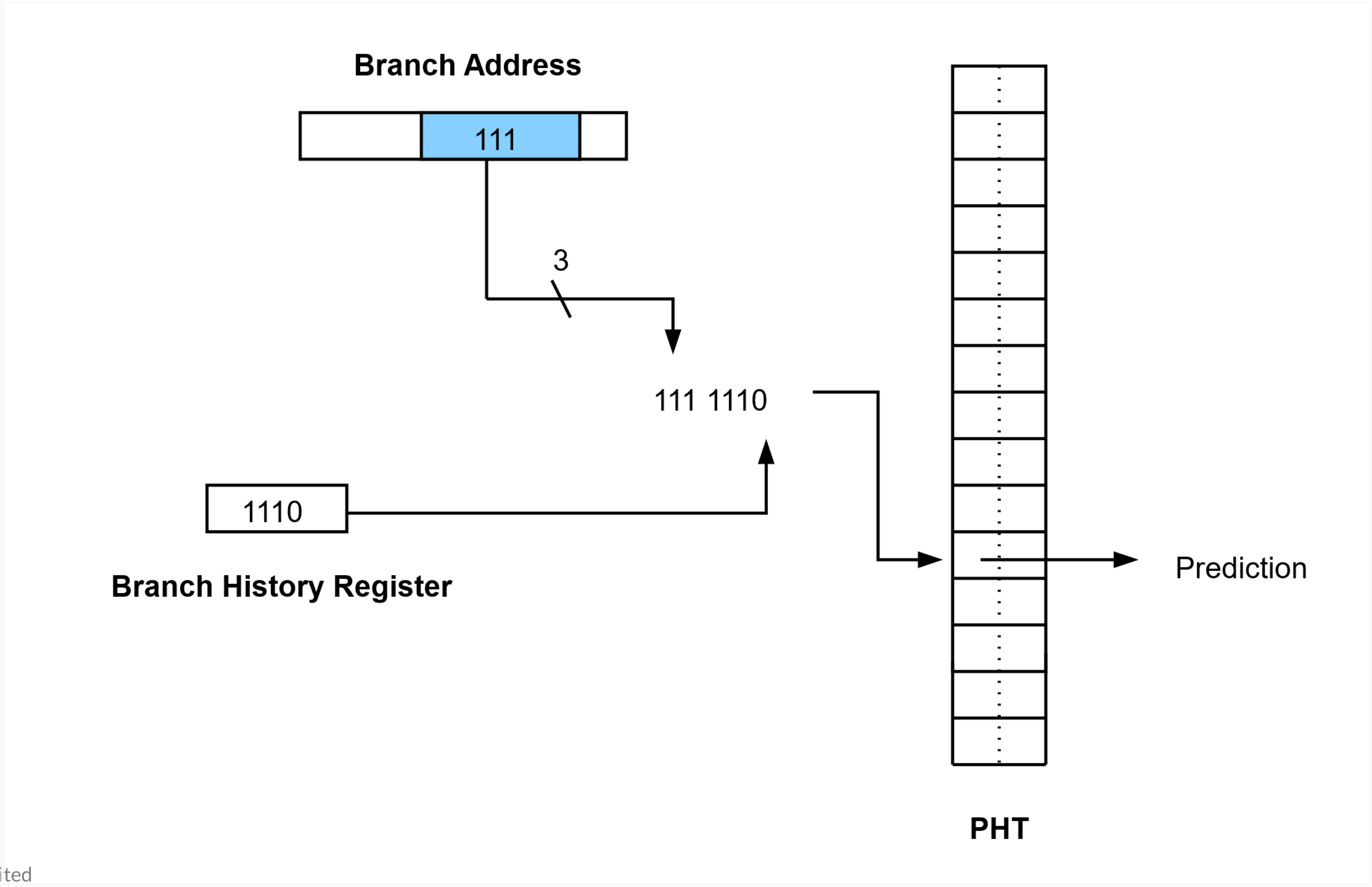


Exploiting Global Branch History

- In addition to exploiting the local history of a particular branch, we can also note that the behavior of a branch is often correlated with the behavior of other recent branches (the global history):

- `If (cond1) { .... }`
- `If (cond2) { .... }`
- `If (cond1 && cond2) { .... }  // dependent on outcome of`
  `// previous branches`

Global History Two-level Predictor



Optimizations

Tournament Predictors

More advanced schemes employ both local and global history predictors

These so-called “tournament predictors” also employ a third predictor (another table of 2-bit counters) to select which predictor’s output (local or global) to use.

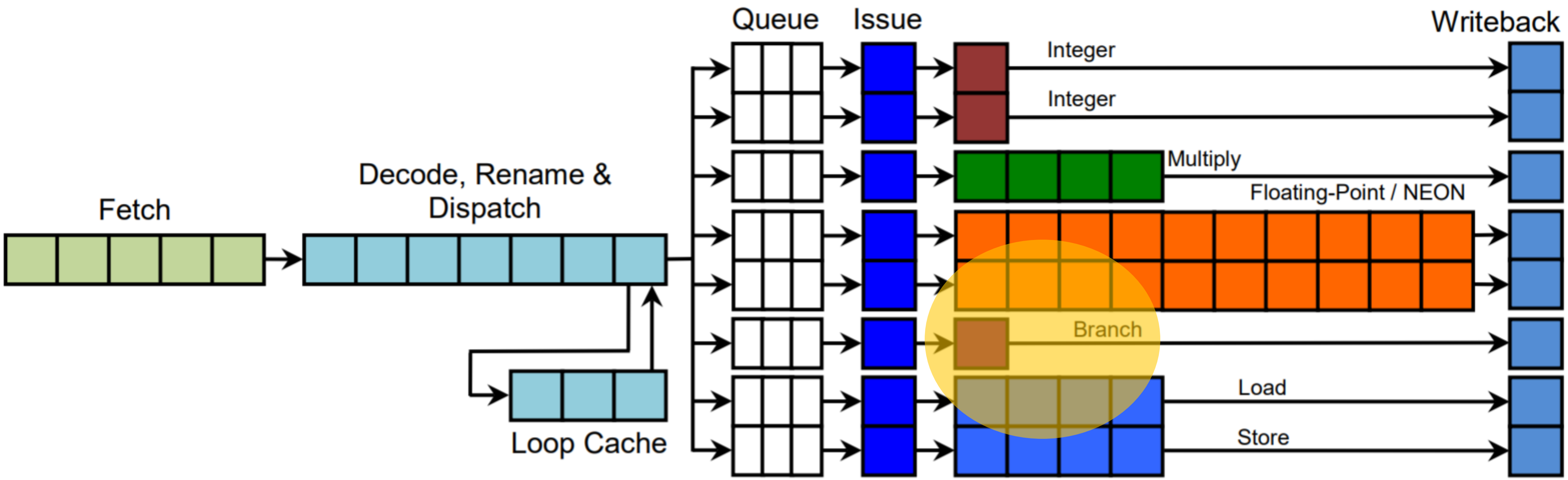
Aliasing problems

Performance may be limited by negative interference, i.e., when two branches map to same entry in PHT but are biased in opposite directions. Numerous schemes try to deal with this, e.g., by using multiple PHTs or using a small tagged “cache” to hold branches that experience interference.

Limits to Dynamic Branch Prediction

- Some branches are unpredictable (dependent on input data).
- Need to “train” predictor for some period before predictions are accurate
- Predictor accuracy will be limited by the area (cost), cycle time, or power of the hardware.
- Aliasing and interference

Example: Arm Cortex-A15



Arm Cortex-A15 pipeline

Example: Arm Cortex-A15

The Cortex A-15 uses a “bi-mode” predictor, an extension of the global history two-level predictor to reduce limitations due to destructive aliasing.

There are now two PHTs, each with 8192 entries and an additional “choice” predictor to choose between the two PHTs.

Overall, the branch predictor is relatively large and consumes ~15% of the core’s power.

Branching Without Stalls

What do we need to know to completely avoid stalling on a branch?

1. Need to know the instruction we are “currently” fetching is a branch (remember, it hasn’t returned from memory yet, so how can we know?)
2. We need to predict the branch taken or not-taken.
3. If the branch is predicted as taken, we will need to know the branch’s target address.

To provide the information to solve issues 1 and 3, we store recent branches together with their target addresses in a **Branch Target Buffer (BTB).**

The Branch Target Buffer (BTB)

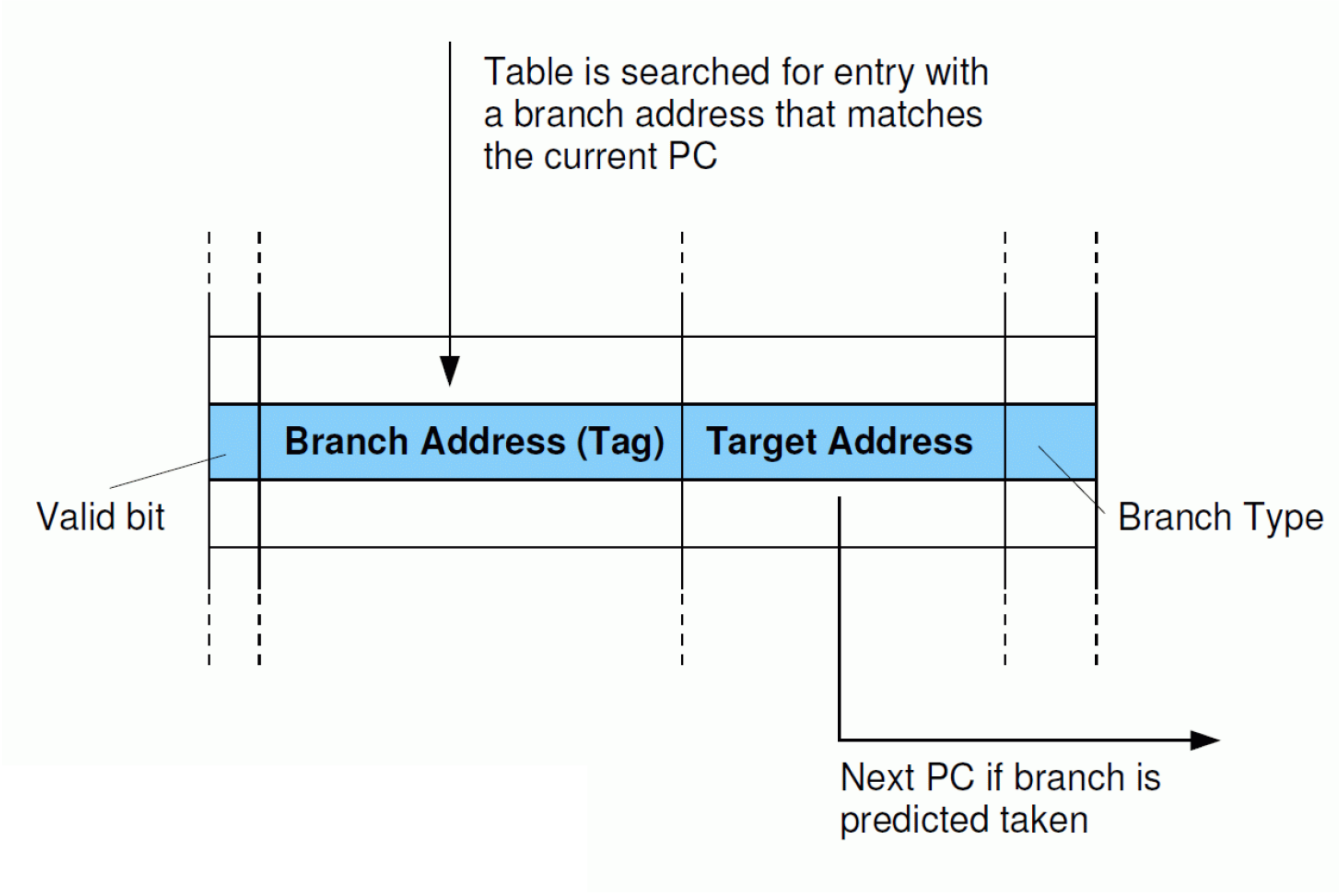
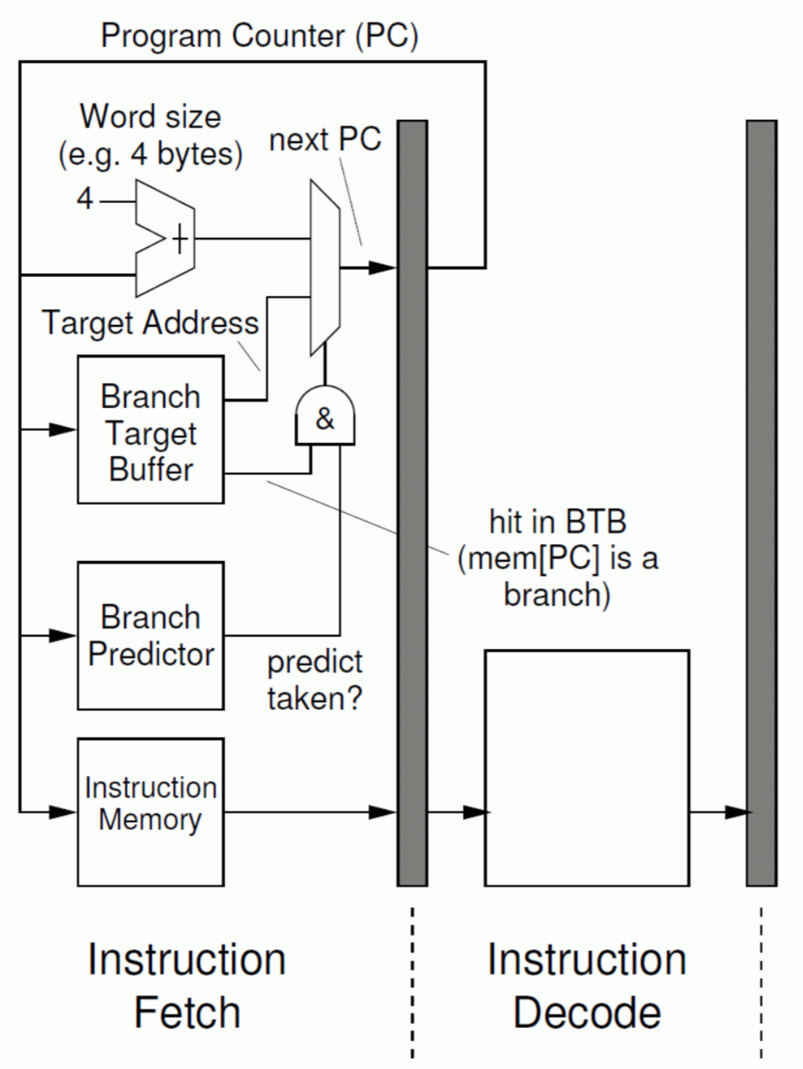


Table is searched for entry with a branch address that matches the current PC
Branch Address (Tag)
Target Address
Valid bit
Branch Type
Next PC if branch is predicted taken

Putting It All Together



Rest of pipeline…

Other BTB Tricks: Branch Folding

- In addition to storing the branch target address, we could store the target instruction in the BTB.
- No need to fetch the next instruction; CPI for branch is effectively zero now.
- Could also allow us to take longer to access (a larger) BTB
- The branch has now been removed from the instruction stream that is presented to the execution pipeline – instead, the branch is substituted for the branch target instruction
- We may also create a separate structure to cache instructions at the branch target (i.e. the **Branch Target Instruction Cache** or **BTIC**)

Return Address Predictors

Functions may be called from multiple places in the program.

The accuracy of the branch target (return address) stored in the BTB may be very low.

Solution: use a small hardware stack to store these addresses (the **return-address stack**).

What if this stack overflows?

Exceptions

Exceptions

- In some situations, we are required to interrupt a program's execution and take some action. These conditions or system events are called *exceptions*. The necessary action is taken by privileged software, i.e., the *exception handler.*
- Exceptions may occur for many different reasons, e.g.,
  - A page fault, breakpoint, I/O device request, floating-point errors, memory protection violation, etc.

Exceptions

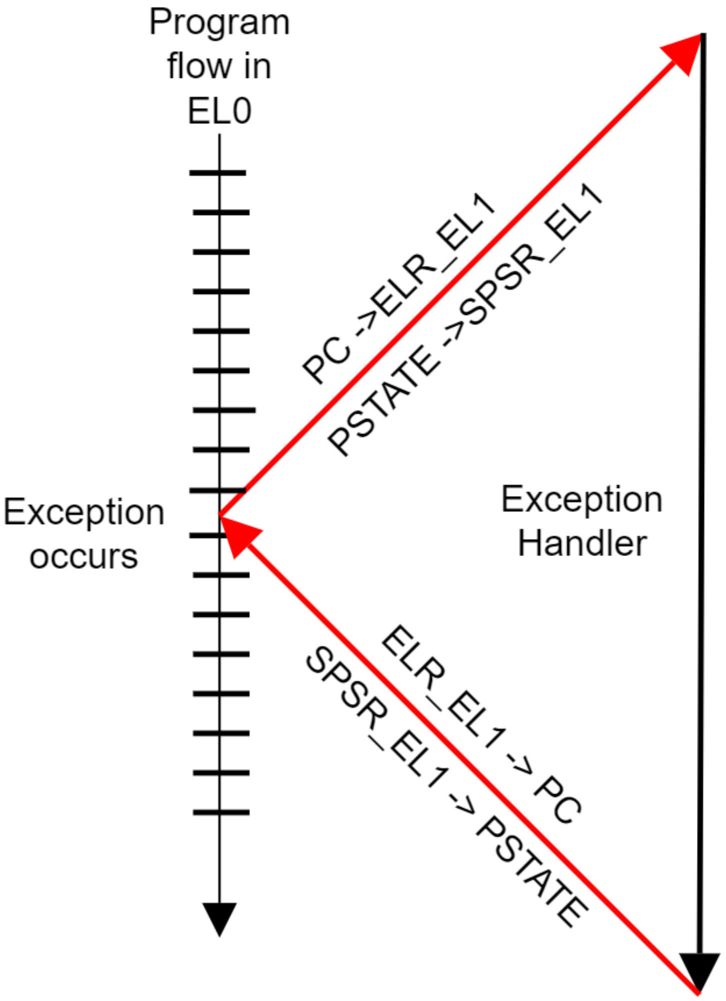
Types of exceptions (for Arm):

- Interrupts
- Aborts
- Reset
- Exception generating instructions

Exceptions

An exception will cause the processor to perform the following:

1. Save the Processor State (PSTATE), e.g., processor flags, interrupt mask bits, exception level, etc.
2. Save the return address (current PC).
3. Branch to handler specified in vector table
4. Save registers, execute handler code, restore registers.
5. Return from exception (ERET instruction).



Exceptions

The intention is to temporarily interrupt program execution, deal with the exception, and then to resume execution.

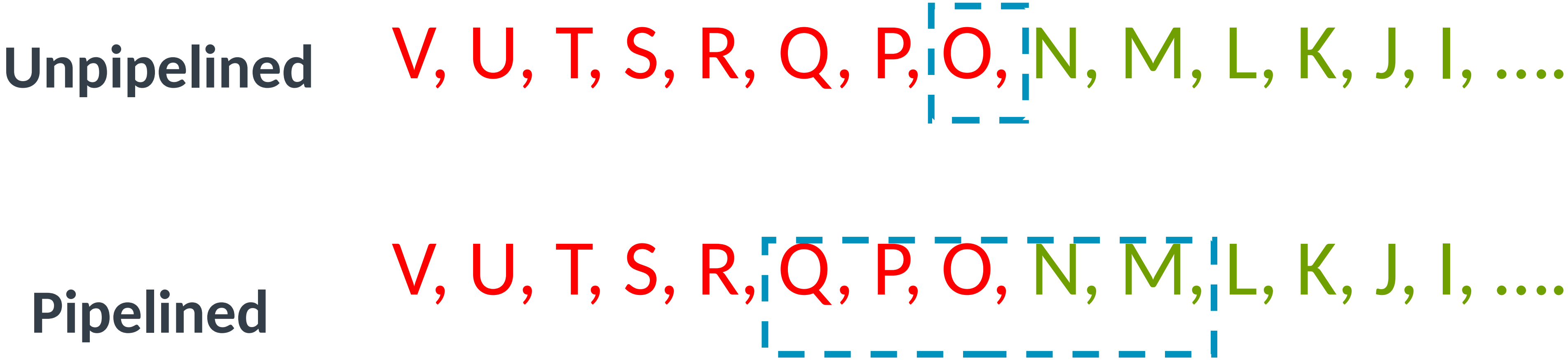
A good way to ensure we can easily resume is to ensure that the architectural state is consistent with the sequential model of program execution before the exception is taken, i.e., if the instruction that causes the exception is instruction E:

1. All instructions prior to E should have completed and updated their destination registers. All exceptions caused by these instructions should have been handled.
2. Any instructions after E in program order should not have completed and not have modified any processor state.
3. Whether E should complete or not will depend on the exception.

These are called “**precise exceptions.**”

Precise Exceptions and Pipelining

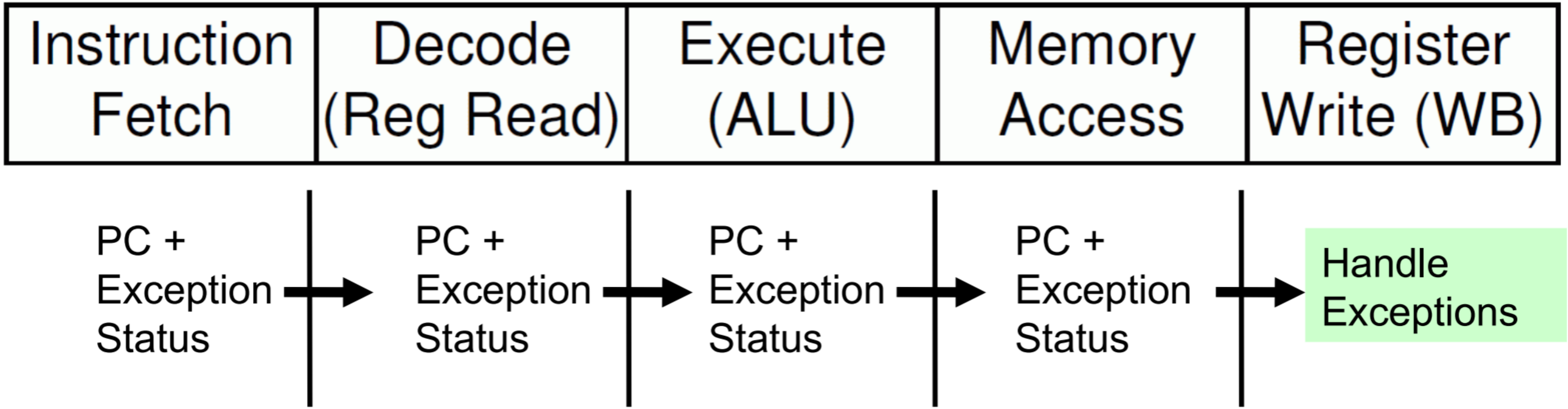
The requirements on the previous slide are trivial in the case of an unpipelined processor, but more complex for a pipelined processor.



Unpipelined V, U, T, S, R, Q, P, O, N, M, L, K, J, I, ….

Pipelined V, U, T, S, R, Q, P, O, N, M, L, K, J, I, ….

Precise Exceptions



Instruction Fetch

Decode (Reg Read)

Execute (ALU)

Memory Access

Register Write (WB)

PC + Exception Status

PC + Exception Status

PC + Exception Status

PC + Exception Status

Handle Exceptions

The Limits of Pipelining

Limits to Pipelining

- As we saw in the last module, a deeper pipeline doesn’t necessarily lead to better performance.
- We need to work hard to feed the pipeline with instructions and data and to minimize pipeline stalls.
- If we pipeline our execute stage, we will also need to find successive independent instructions to keep our pipeline from stalling.

Limits to Pipelining

Pipelining is also ultimately limited by lower-level concerns:

- Register and clocking overheads are non-zero. If we have very little logic per pipeline stage, these may represent a significant fraction of our critical path delay.
- Need to balance logic between pipeline stages. Clock period is determined by worst-case delay.
- Limits on number of pipelining registers

Limits to Pipelining

- Ultimately, we will be unable to increase the performance of a processor that only attempts to issue a single instruction per clock cycle.
- Even before this point is reached, it is preferable to use multiple-issue techniques, i.e., do more work in each pipeline stage: fetch and execute multiple instructions per clock cycle.
- Details in the next module