14. Basic Processor Design – Pipelining With Control Hazards

EECS 370 – Introduction to Computer Organization – Winter 2023

EECS Department
University of Michigan in Ann Arbor, USA

Due dates

- Project 2L is due this Thursday (2/23)

- Midterm is Thursday 3/9, 7-9pm.
  - Practice exams posted on Piazza (@1577)
  - Exam covers through this Thursday's lecture.
    - Most practice exams don’t cover pipelining because our exam is a bit later (after the break) than most terms.

Week 7: February 21 - February 27

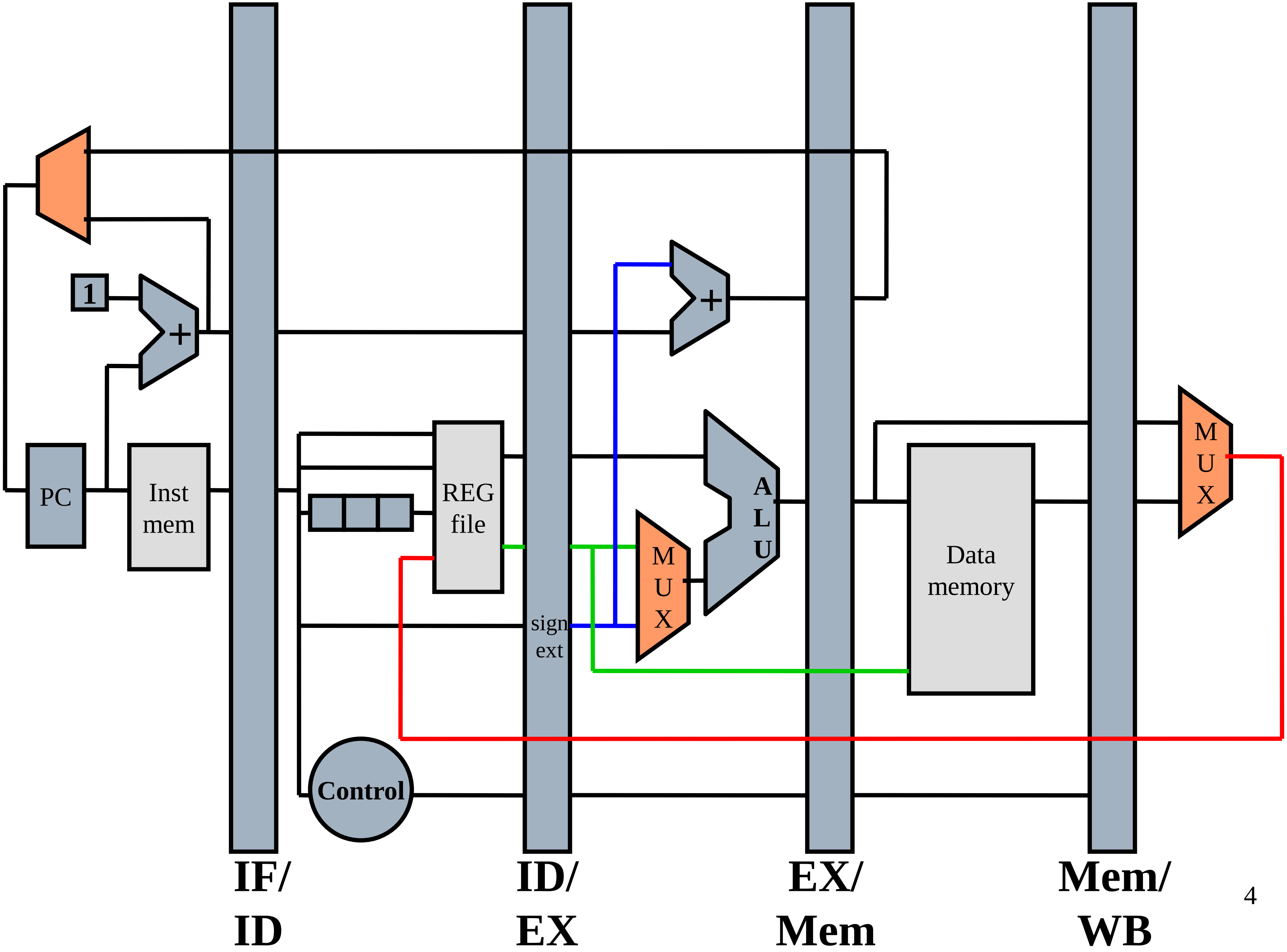
Tue February 21	Lecture 14: Data and Control Hazards Slides			Sections 4.7, 4.8
Thurs February 23	Lecture 15: Control Hazards and Exceptions Slides		Project 2.l	Sections 4.7, 4.8, 4.9
Fri February 24	No Discussion			
Mon February 27				

Break: February 28 - March 6

Week 8: March 7 - March 13

Tue March 7	Lecture 16: Midterm review Slides Youtube	Project 3 && HW 4		Sections 5.1, 5.2, 5.3
Thur March 9	Midterm, 7-9pm ET			
Fri March 10	Discussion 7: Project 3: Pipelining Slides, Worksheet, Studio Recording			
Mon March 13				

Review: LC2k Pipelined Datapath



What can go wrong?

- Data hazards: since register reads occur in stage 2 and register writes occur in stage 5 it is possible to read the wrong value if is about to be written.

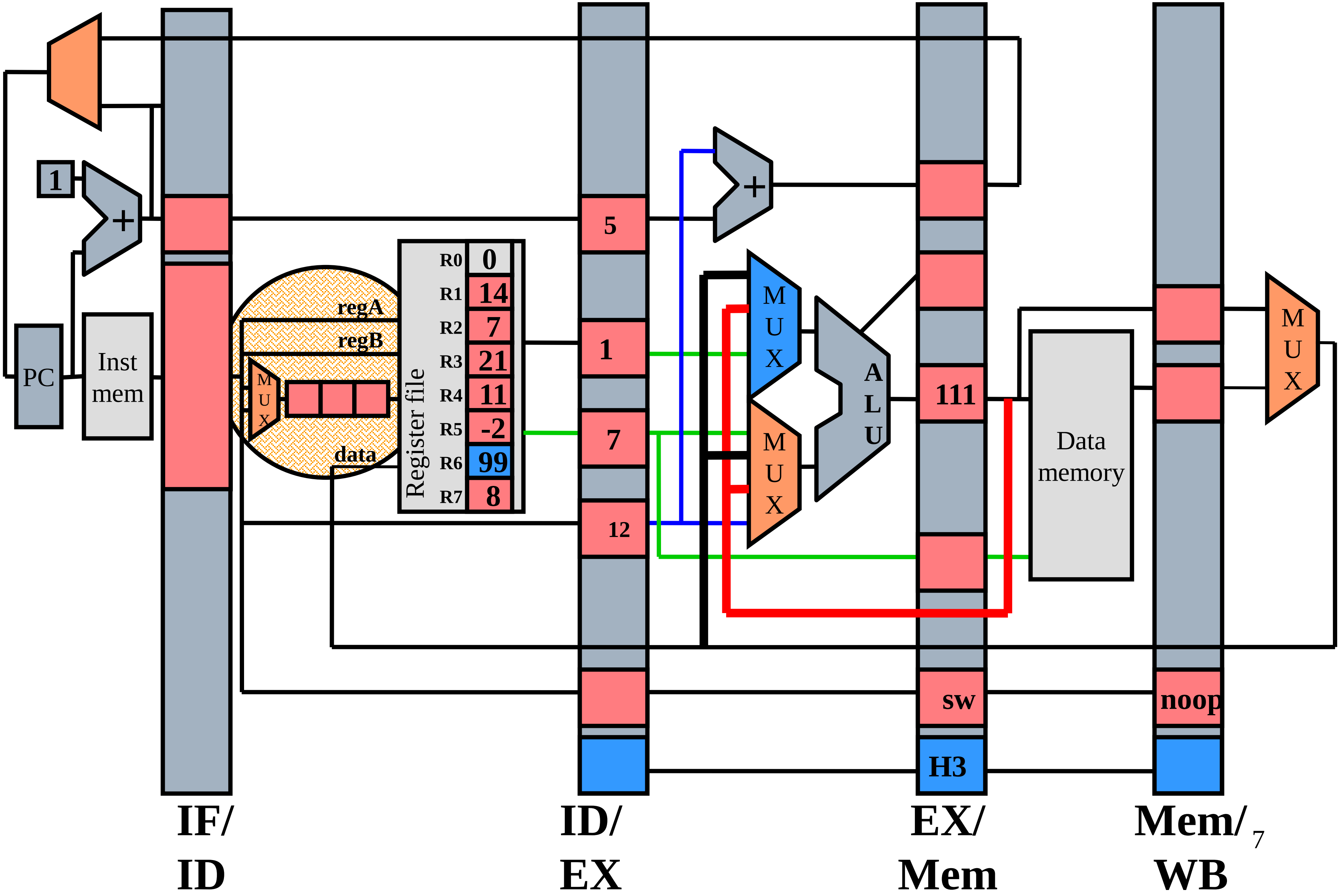
- Control hazards: A branch instruction may change the PC, but not until stage 4. What do we fetch before that?

- Exceptions: How do you handle exceptions in a pipelined processor with 5 instructions in flight?

Review: approaches to handling data hazards

- Avoid
  - Make sure there are no hazards in the code.
- Detect and stall
  - If hazards exist, stall the processor until they go away.
- Detect and forward
  - If hazards exist, fix up the pipeline to get the correct value (if possible).

Data forwarding



Example time graph

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
add 1 2 3	IF	ID	EX	ME	WB								
nor 3 4 5		IF	ID	EX	ME	WB							
add 6 3 7			IF	ID	EX	ME	WB						
lw 3 6 10				IF	ID	EX	ME	WB					
sw 6 2 12					IF	ID*	ID	EX	ME	WB			

Project 3 design tips

- Build up your simulator in pieces.
  - First, design code without any hazards – get the pipeline flow to work for all instructions
  - Build up your data forwarding (remember forwarding from 3 places back to EX: EX/MEM, MEM/WB, and WB/END).
  - Handle control hazards.
  - One extra cycle for stalls (simple register file implementation).
- Testcases are critical to debugging your code – don't rely on the autograder!
- Use your functionally correct previous "golden design" for testing.
- Implement without considering hazards first, test with hazard-free code, then consider hazards.
- Go to discussion.

Control hazards

- How can the pipeline handle branch and jump instructions?

Branch
Prediction

Pipeline function for BEQ

- Fetch: read instruction from memory.
- Decode: read source operands from registers.
- Execute: calculate target address and test for equality.
- Memory: **Send target to PC** if test is equal.
- Writeback: Nothing left to do.
- **Branch outcomes**
  - **Not Taken**
    - PC = PC + 1
  - **Taken**
    - PC = Branch Target Address

Control hazards

beq 1 1 10
add 3 4 5

time →

beq	**fetch**	**decode**	**execute**	**<u>memory</u>**	**writeback**
add		**<u>fetch</u>**	**decode**	**execute**	

Approaches to handling control hazards

- Avoid
  - Make sure there are no hazards in the code.
- Detect and stall
  - Delay fetch until branch resolved.
- Speculate and Squash-if-Wrong
  - Go ahead and fetch more instructions in case it is correct, but stop them if they shouldn't have been executed.

Handling control hazards I: Avoid all hazards

- **Don’t have branch instructions!**
  - Impractical.

- **Delay taking branch**
  - dbeq r1 r2 offset
  - Instructions at PC+1, PC+2,…, PC + <# delay slots> will execute before deciding whether to fetch from PC+1+offset.
  - If no useful instructions can be placed after dbeq, noops must be inserted.

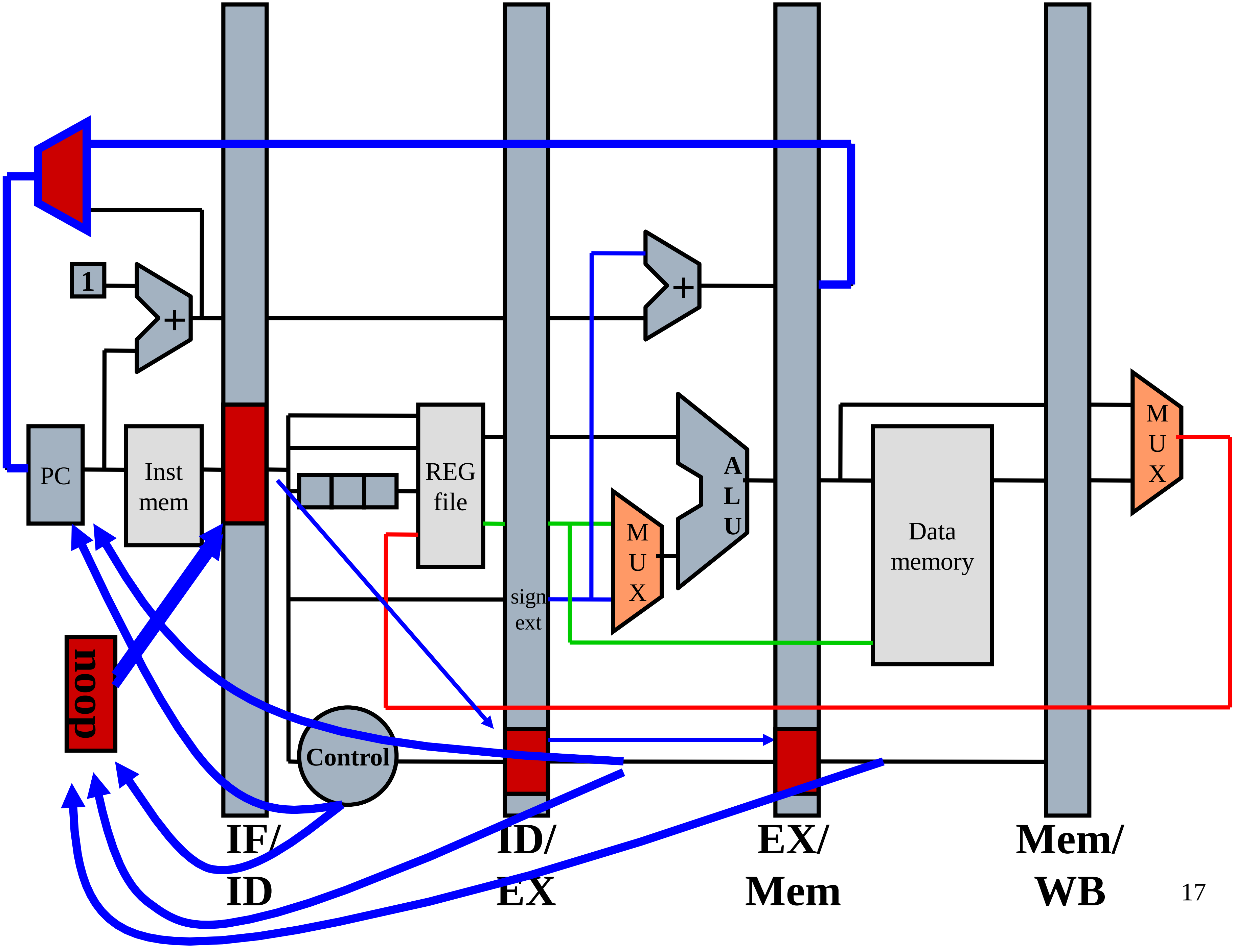
Problems with delayed branches

- Old programs (legacy code) may not run correctly on new implementations.
  - Longer pipelines need more instructions/noops after delayed beq.
- Programs get larger as noops are included.
  - Especially a problem for machines that try to execute more than one instruction every cycle.
  - Intel EPIC: Often 25%–40% of instructions are noops.
- Program execution is slower.
  - CPI equals 1, but some instructions are noops.

Handling control hazards II: detect and stall

- Detection
  - Must wait until decode.
  - Compare opcode to beq or jalr.
  - Alternately, this is just another control signal.

- Stall
  - Keep current instructions in fetch.
  - Pass noop to decode stage, not execute!



1
+
PC
Inst mem
REG file
sign ext
+
M U X
A L U
Data memory
M U X
Control
noop
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

Control hazards

beq 1 1 10
add 3 4 5

time

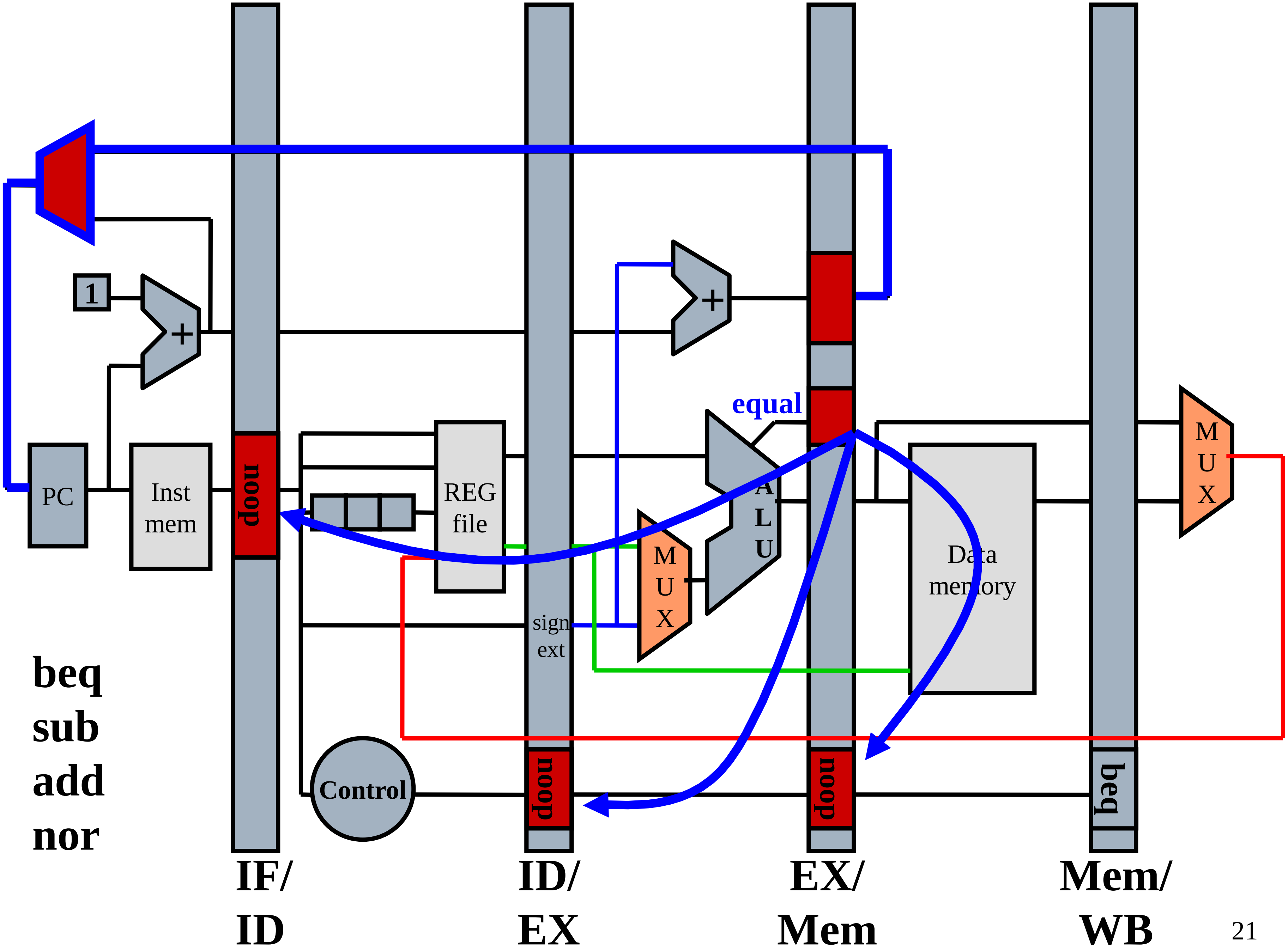
beq	fetch	decode	execute	<u>memory</u>	writeback
add		fetch	<u>fetch</u>	<u>fetch</u>	fetch
					or
Target:					fetch

Problems with detect and stall

- CPI increases every time a branch is detected!
- Is that necessary? Not always!
  - Branch not always taken.
  - Let's assume that it is NOT taken…
    - In this case, we can ignore the beq (treat it like a noop).
    - Keep fetching PC + 1.
  - What if we are wrong?
    - OK, as long as we do not COMPLETE any instructions we mistakenly executed.
    - I.e., make changes that will be seen later such as changing register or memory values.

Handling control hazards III: Speculate and squash

- Speculate: assume not equal
  - Keep fetching from PC+1 until we know that the branch is really taken.
- Squash: stop bad instructions if taken
  - Send a noop to Decode, Execute, and Memory.
  - Send target address to PC.

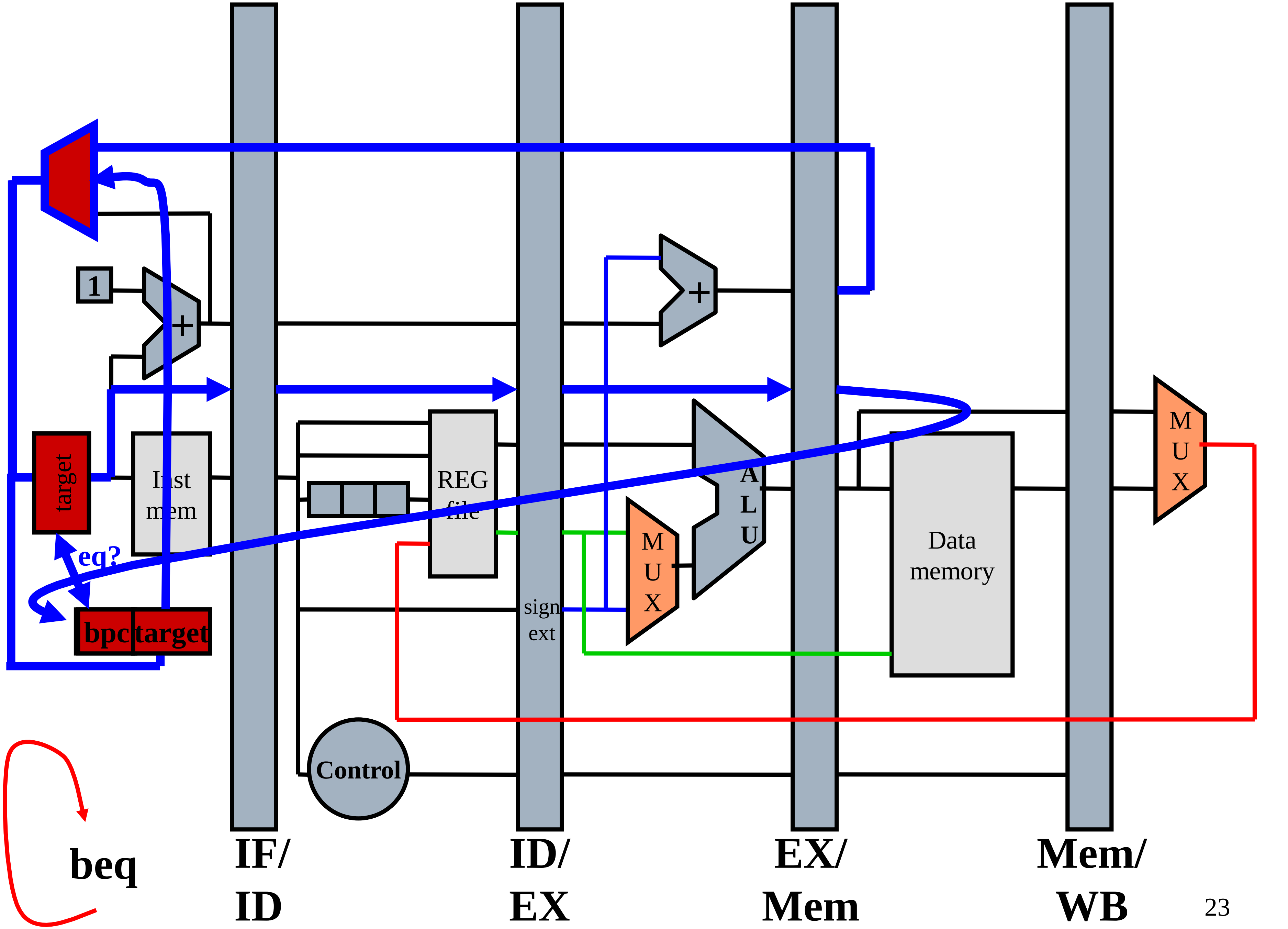


1
+
+
PC
Inst
mem
noop
REG
file
equal
A
L
U
M
U
X
sign
ext
Data
memory
M
U
X
Control
noop
noop
beq
beq
sub
add
nor
IF/
ID
ID/
EX
EX/
Mem
Mem/
WB

Problems with fetching PC+1

- CPI increases every time a branch is taken!
  - About 50%-66% of time.
- Is that necessary?

No! But how can you fetch from the target before you even know the previous instruction is a branch – much less whether it is taken?


target
1
+
Inst
mem
eq?
bpc
target
REG
file
Control
sign
ext
+
M
U
X
A
L
U
Data
memory
M
U
X
beq
IF/
ID
ID/
EX
EX/
Mem
Mem/
WB

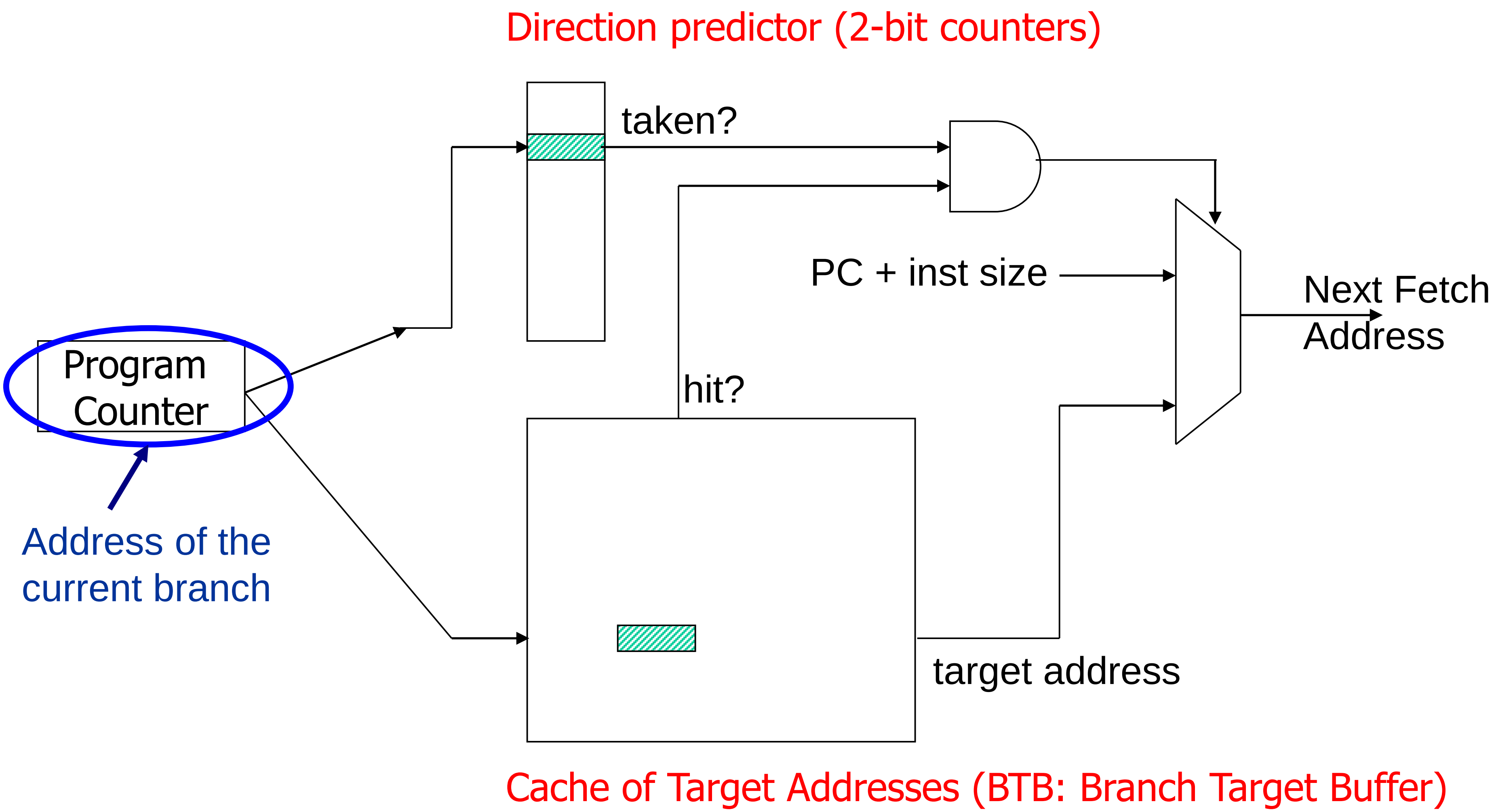
Branch Prediction

- Predict the next fetch address (to be used in the next cycle)

- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - Branch direction (if conditional)
  - Branch target address (if direction is taken)

- Observation: Target address remains the same for a conditional direct branch across dynamic instances
  - Store the target address from previous instance and access it with the PC
  - Called Branch Target Buffer (BTB) or Branch Target Address Cache

Fetch Stage with Branch Prediction



Branch Direction Prediction

Branch Direction Prediction

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid

Branch Direction Prediction (Static)

- Always not-taken
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40%
  - Compiler can layout code such that the likely path is the “not-taken” path
- Always taken
  - No direction prediction
  - Better accuracy: ~60-70%
    - Backward branches (i.e. loop branches) are usually taken
    - Backward branch: target address lower than branch PC
- Backward taken, forward not taken (BTFN)
  - Predict backward (loop) branches as taken, others not-taken

Branch Direction Prediction (Dynamic)

- Last time predictor
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed
    - TTTTTTTTTTNNNNNNNNNN → 90% accuracy

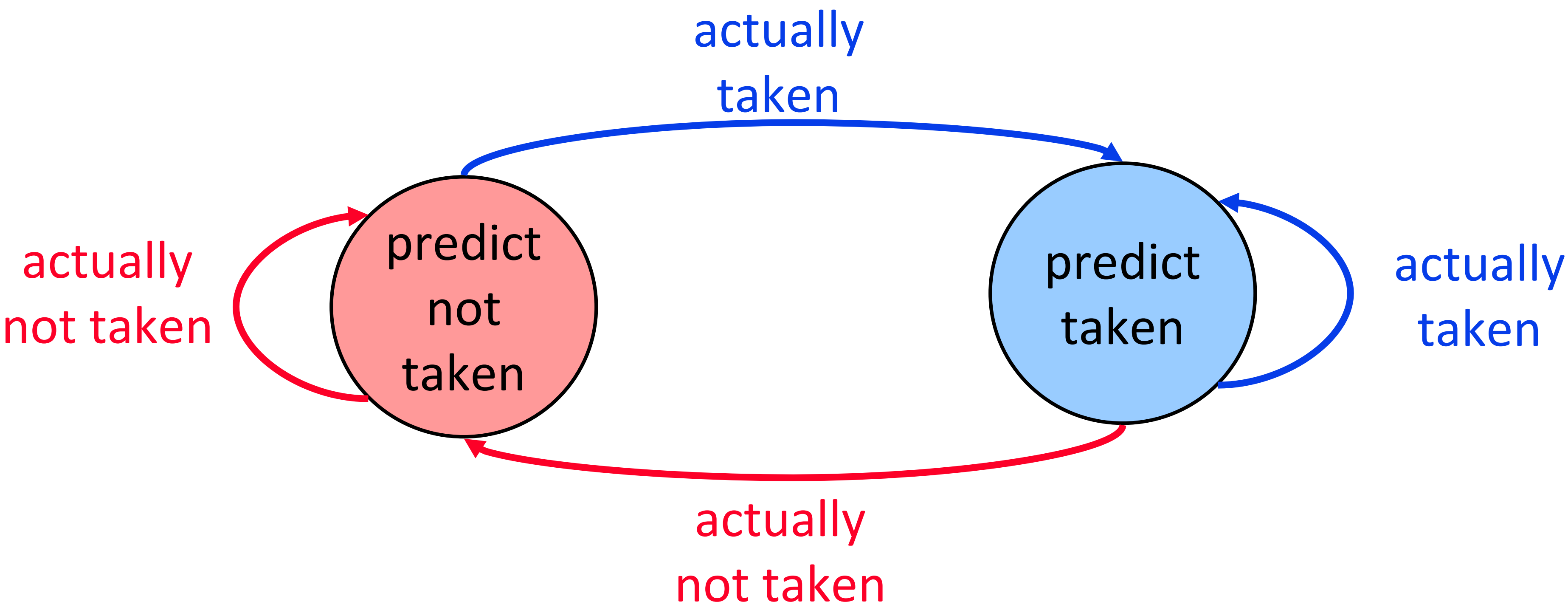
- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N-2)/N

+ Loop branches for loops with large number of iterations

-- Loop branches for loops will small number of iterations

TNTNTNTNTNTNTNTNTNTN → 0% accuracy

State Machine for Last-Time Prediction



Improving the Last Time Predictor

- Problem: A last-time predictor changes its prediction from T→NT or NT→T too quickly
  - Even though the branch may be mostly taken or mostly not taken

- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
  - Use two bits to track the history of predictions for a branch instead of a single bit
  - Can have 2 states for T or NT instead of 1 state for each

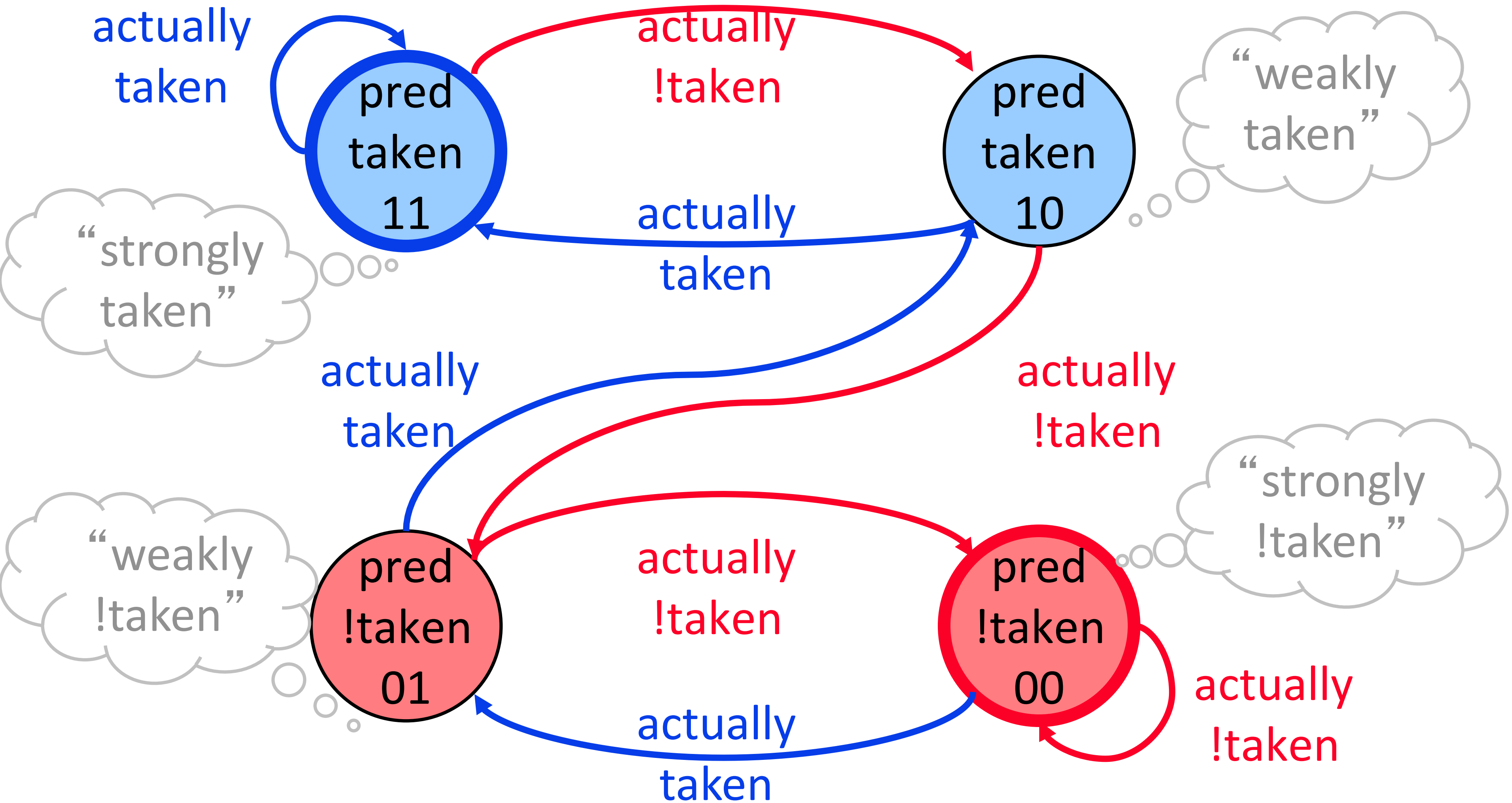
Two-Bit Counter Based Prediction

- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome

\+ Better prediction accuracy

-- More hardware cost (but counter can be part of a BTB entry)

State Machine for 2-bit Saturating Counter



Two-Bit Counter Based Prediction

- What's the prediction accuracy of a branch with the following sequence of taken/not taken evaluations
  - T T T T N T T N N N T N T N N

Br	T	T	T	T	N	T	T	N	N	N	T	N	T	N	N
State	10	11	11	11	[illegible]	10	11	[illegible]	[illegible]	01	[illegible]	01	[illegible]	01	00
Pred	T	T	T	T	T	T	T	T	T	N	N	N	N	N	N

Remember this Example from Lecture 1?

We know understand why sorting improves the inner-loop so much

The branch predictor is better at guessing what's gonna happen when data is sorted!

```
for (unsigned c = 0; c < arraySize; ++c)
    data[c] = std::rand() % 256;
std::sort(data, data + arraySize);

// Test
clock_t start = clock();
long long sum = 0;
// Primary loop
for (unsigned c = 0; c < arraySize; ++c)
{
    if (data[c] >= 128)
        sum += data[c];
}

double elapsedTime =
  static_cast<double>(clock() - start);
```

Branch prediction

Predict not taken:	~50% accurate.
Predict backward taken:	~65% accurate.
Predict same as last time:	~80% accurate.
Pentium:	~85% accurate.
Pentium Pro:	~92% accurate.
Best paper designs:	~96% accurate.

Can We Do Better?

Last-time and 2BC predictors exploit "last-time" predictability

Realization 1: A branch's outcome can be correlated with other branches' outcomes

Global branch correlation

Realization 2: A branch's outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch "last-time" it was executed)

Local branch correlation

Global Branch Correlation

Recently executed branch outcomes in the execution path is correlated with the outcome of the next branch

```
if (cond1)
...
if (cond1 AND cond2)
```

If first branch not taken, second also not taken

```
branch Y: if (cond1) a = 2;
...
branch X: if (a == 0)
```

```
if (x<1) . . .
if (x>1) . . .
```

If first branch taken, second definitely not taken

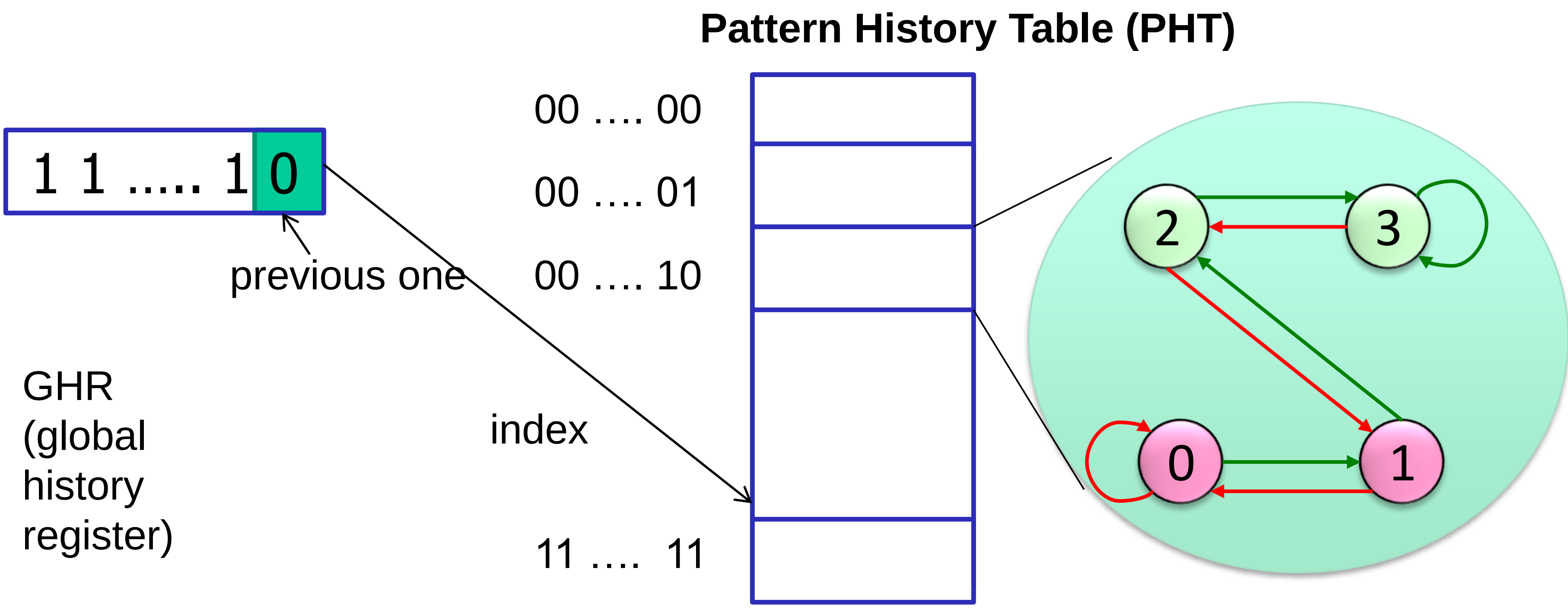
Two Level Global Branch Prediction

First level: Global branch history register (N bits)

The direction of last N branches

Second level: Table of saturating counters for each history entry

The direction the branch took the last time the same history was seen



Can We Do Better?

Last-time and 2BC predictors exploit "last-time" predictability

Realization 1: A branch's outcome can be correlated with other branches' outcomes

Global branch correlation

Realization 2: A branch's outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch "last-time" it was executed)

Local branch correlation

Local Branch Correlation

for (i=1; i<=4; i++) { }

If the loop test is done at the end of the body, the corresponding branch will execute the pattern $(1110)^n$, where 1 and 0 represent taken and not taken respectively, and n is the number of times the loop is executed. Clearly, if we knew the direction this branch had gone on the previous three executions, then we could always be able to predict the next branch direction.

McFarling, "Combining Branch Predictors," DEC WRL TR 1993.

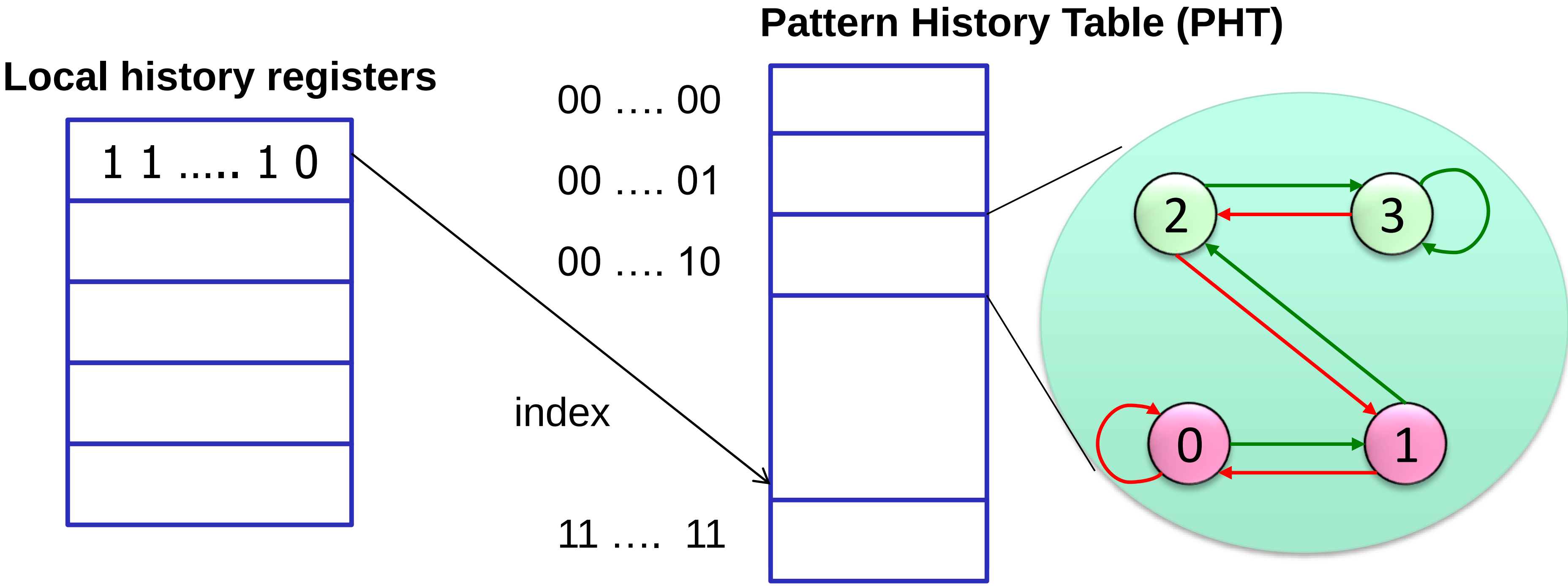
Two Level Local Branch Prediction

First level: A set of local history registers (N bits each)

Select the history register based on the PC of the branch

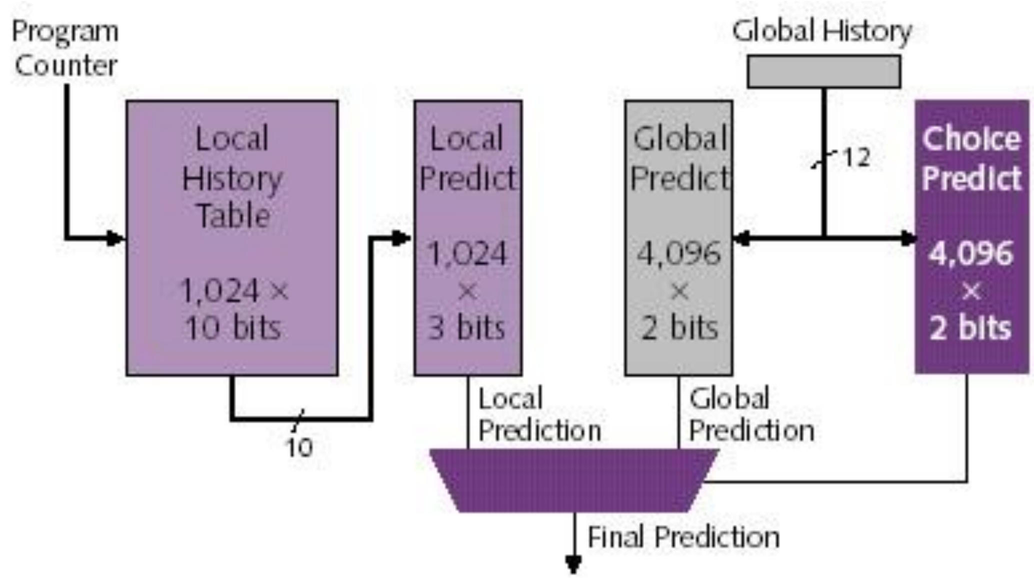
Second level: Table of saturating counters for each history entry

The direction the branch took the last time the same history was seen



What Do Architects Do For Fun?

Alpha 21264 Tournament Predictor



- Minimum branch penalty: 7 cycles
- Typical branch penalty: 11+ cycles
- 48K bits of target addresses stored in I-cache
- Predictor tables are reset on a context switch

Data forwarding – lecture vs. Project 3

- Some questions you may have after you see the P3 spec:
  - What is the WBEND pipeline register in the project for?
  - Why are 3 nops required to avoid hazards in the project?
  - But only 2 nops in class?
- Answer
  - The “magic” register file
    - Lecture register file assumes internal forwarding – in a single cycle, values written to a register are immediately reflected to any reads that occur in the same cycle.
    - Project register file does not do internal forwarding.
  - Most modern processors have internal forwarding as its cheaper than having an additional pipeline register.