13. Pipeline control hazards

EECS 370 – Introduction to Computer Organization

Fall 2013

Profs. Valeria Bertacco, Robert Dick, and Satish Narayanasamy

EECS Department
University of Michigan in Ann Arbor, USA

Announcements

- 5 November: Homework 5 due.
- 12 November: Project 3 due.

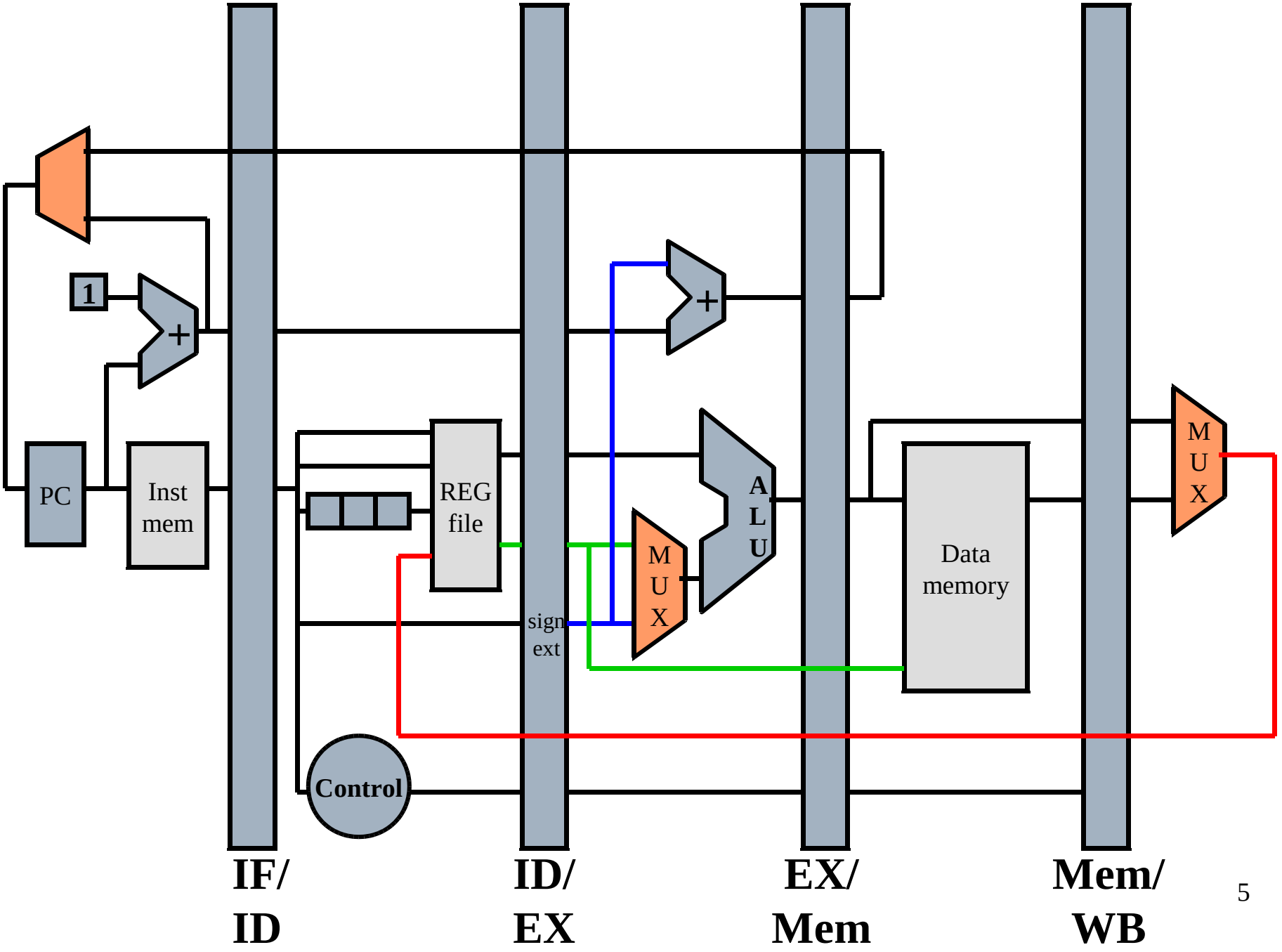
Data forwarding – lecture vs. project 3

- Some questions you may have
  - 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.

Project 3 design tips

- 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.

Review: LC2k Pipelined Datapath



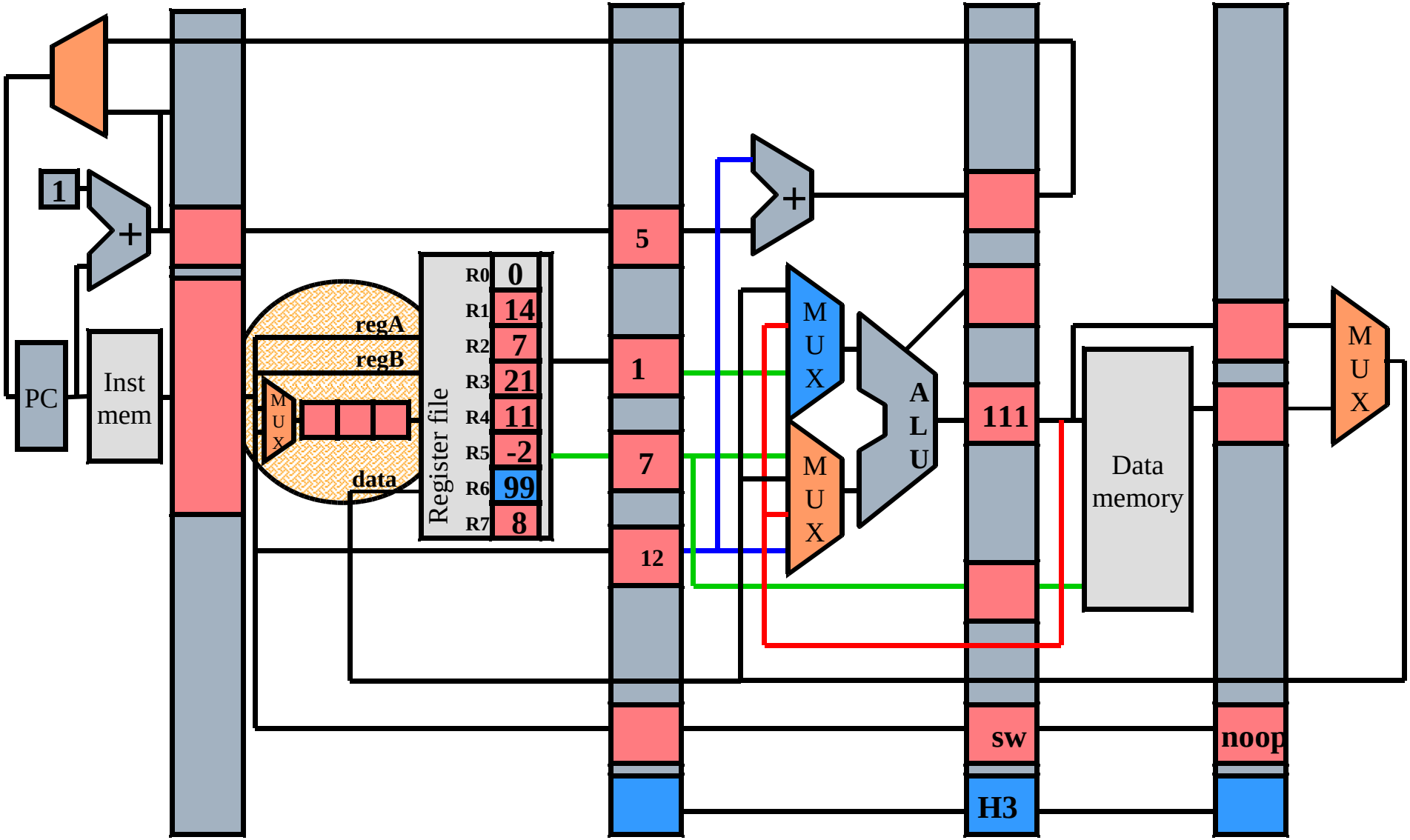
What can go wrong?

- 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?

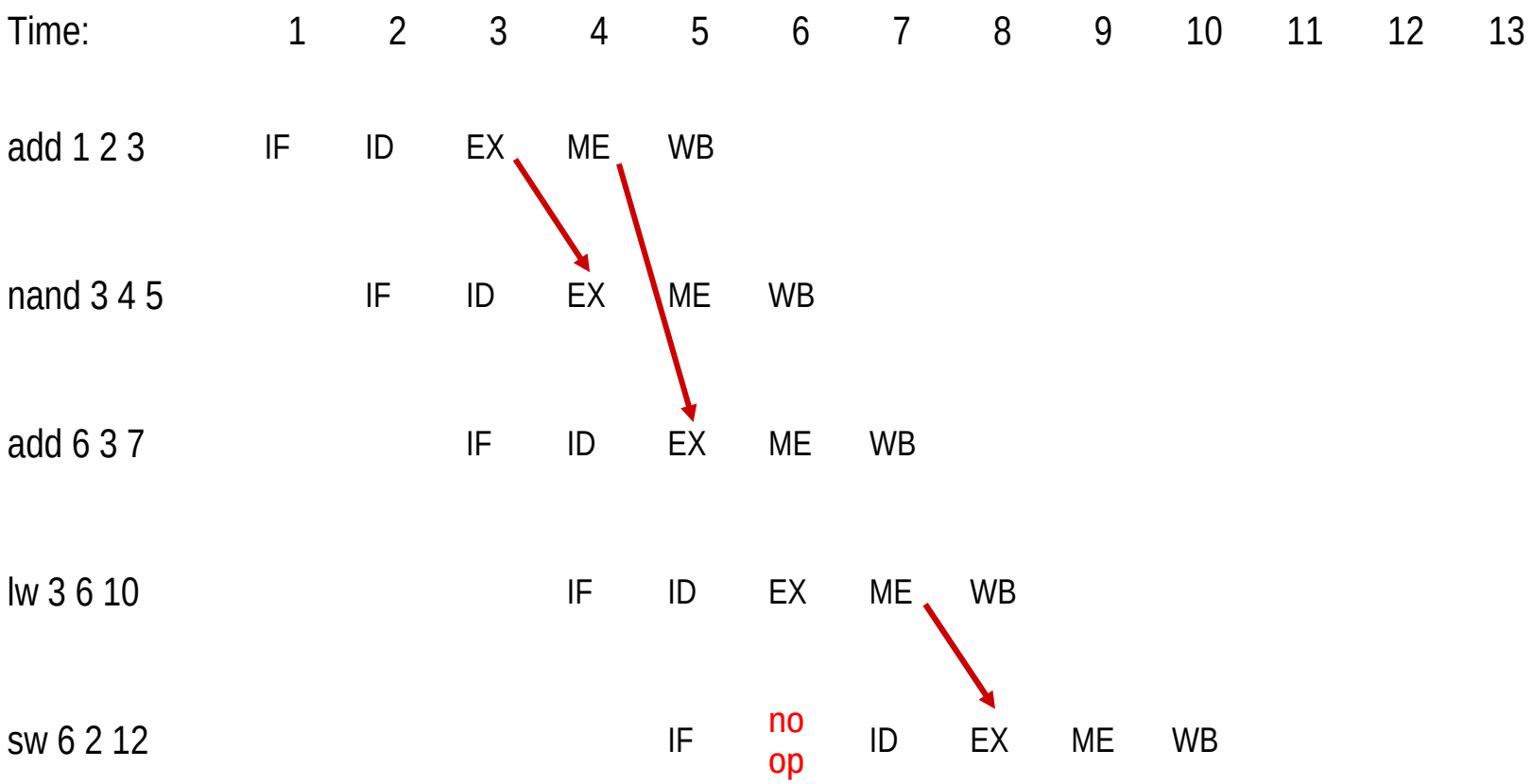
Review: approaches to handling data hazards

- 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



Control hazards

- How can the pipeline handle branch and jump instructions?

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.

Control hazards

beq 1 1 10
sub 3 4 5

beq	**fetch**	**decode**	**execute**	**memory**	**writeback**
sub		**fetch**	**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.

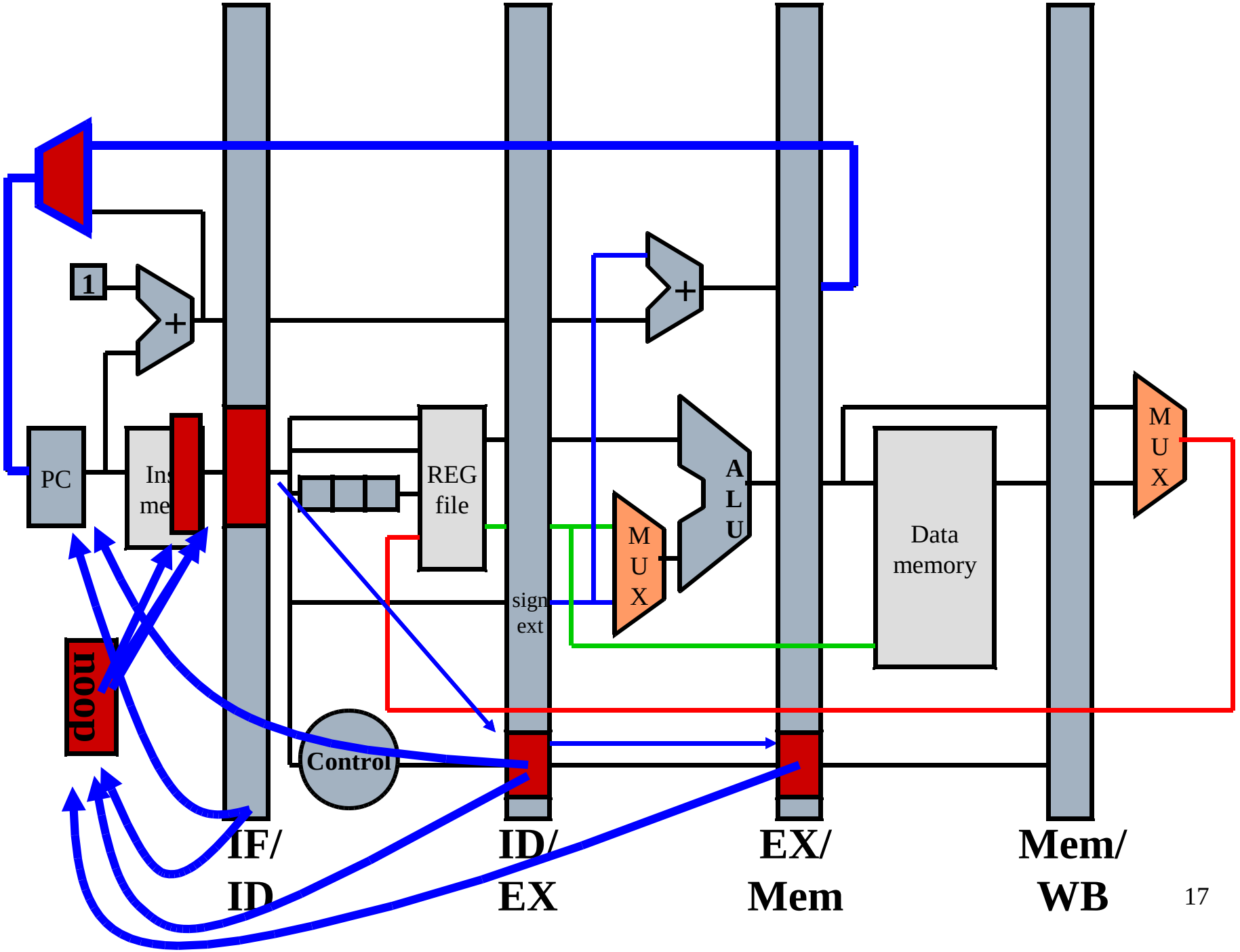
Problems with delayed branches

- 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
Ins
me
REG
file
M
U
X
A
L
U
sign
ext
Data
memory
M
U
X
noop
Control
IF/
ID
ID/
EX
EX/
Mem
Mem/
WB

Control hazards

beq 1 1 10
sub 3 4 5

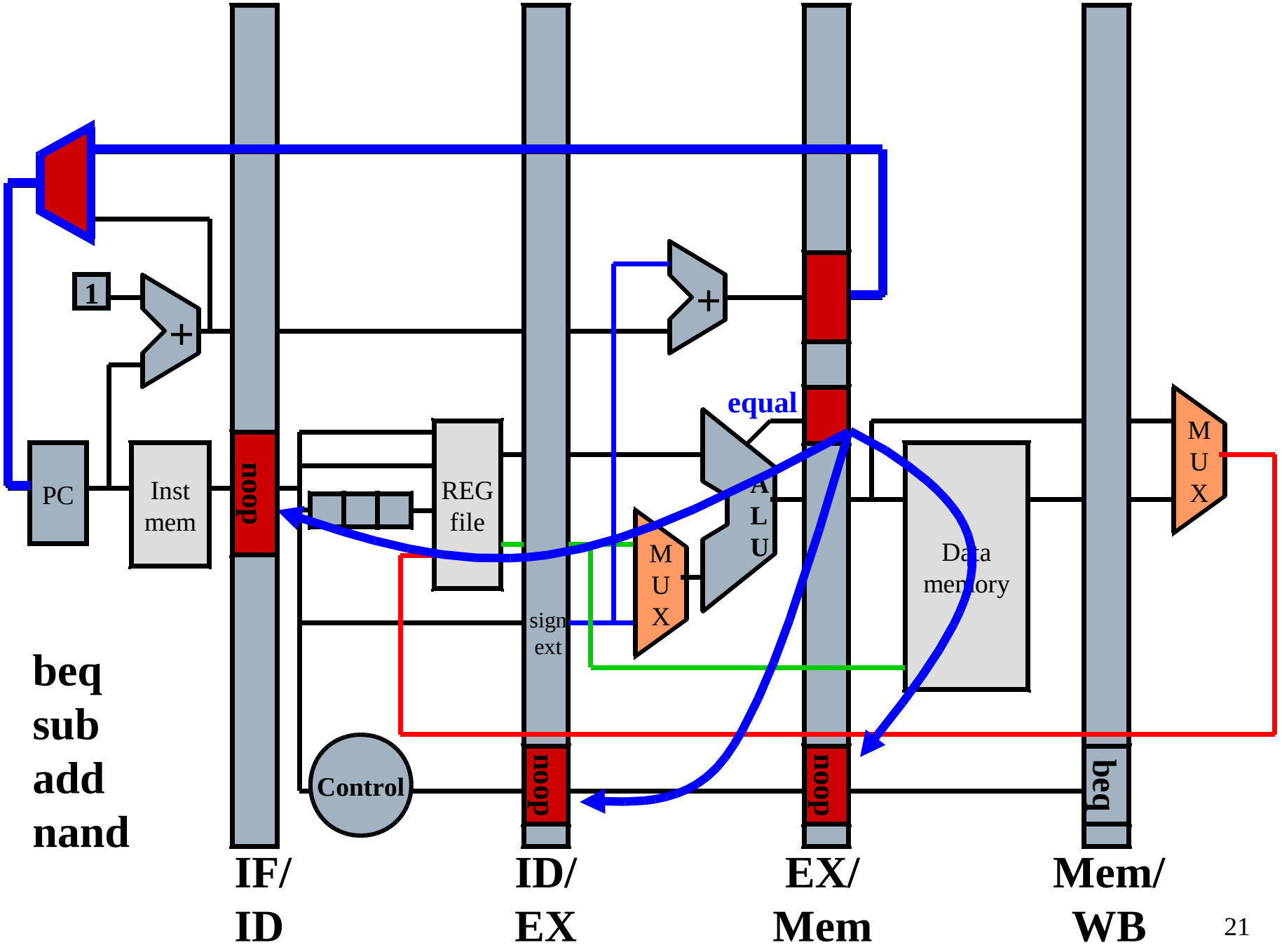
beq	**fetch**	**decode**	**execute**	**memory**	**writeback**
sub		**fetch**	**fetch**	**fetch**	**fetch**
					or
Target:					**fetch**

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

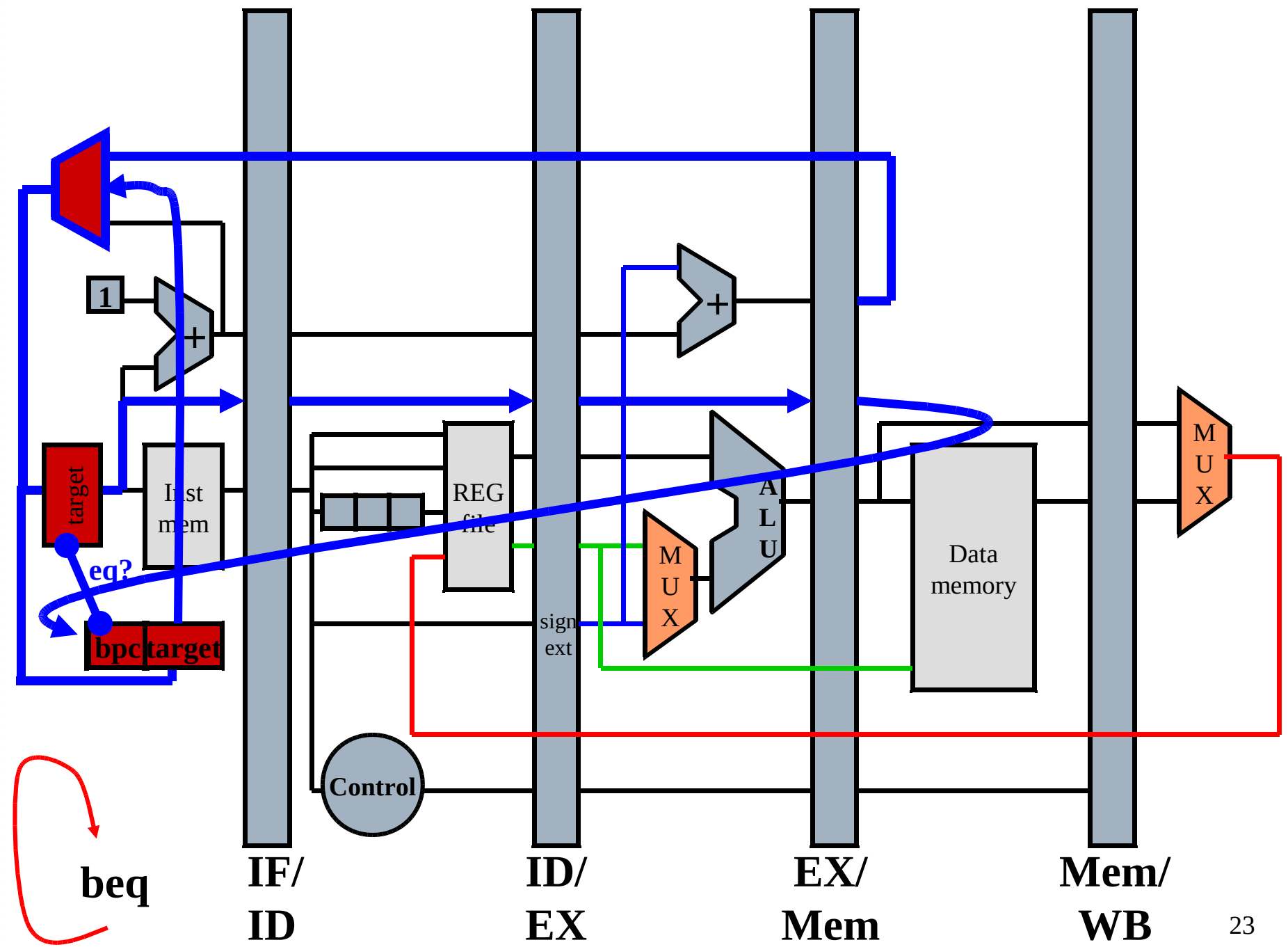
- Speculate: assume not equal
  - Keep fetching from PC+1 until we know that the branch is really taken.
- Squash: stop bad instructions if taken
  - Send a noop to Decode, Execute, and Memory.
  - Send target address to PC.


1
+
+
PC
Inst mem
noop
REG file
sign ext
M U X
A L U
equal
Data memory
M U X
Control
noop
noop
beq
beq
sub
add
nand
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?


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

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.

Branch prediction – a simple approach

- Branch target buffer
  - Table of target addresses for previously executed branches.
  - Store “taken” address.
  - Fallthrough address just PC+1.
- 2-bit saturating counter for each entry
  - Each time branch is taken, increment counter (saturates at 3).
  - Each time branch is not taken, decrement the counter (saturates at 0).
  - Predict direction of branch based on value of counter.
    - Values of 0 or 1, predict not taken.
    - Values of 2 or 3, predict taken.
  - 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 N

Performance impact

Execution time (Time/Program) =

# of instr (I/P) $\times$ CPI (C/I) $\times$ cycle time (T/C)

Multi-cycle decreases cycle time, but increases CPI.

Pipelining decreases CPI.

Down to 1.0 if no stalls (hazards that are fixed by stalling).

Calculating performance with no stalls

```
add   1 2 3
nand  1 4 5
add   4 6 7
```

How many cycles does this code take to execute?

What value is written to the ALU result field of the Mem/WB pipeline register at the end of cycle 5.

The first add is written to the IF/ID pipeline register at cycle time 1.

Calculating performance with data hazards (detect and stall)

```
add   1 2 3
nand  3 4 5
add   3 5 6
```

How many data hazards are there in this code?

How many stall cycles if we use detect and stall to handle the hazards?

Calculating performance with data hazards (detect and forward)

```
add   1 2 3
nand  3 4 5
add   3 5 6
lw    3 6 7
add   6 6 1
```

Where do the values for the second add instruction come from?

How many stall cycles on the LC2K pipelined datapath with data forwarding from lecture?

Calculating performance with control hazards (speculate and squash)

- How many cycles are saved if you perform speculate and squash for the following code (assume that branches are predicted to be not taken)?

```
add   1 2 3
beq   1 5 1
nand  6 4 1
add   3 4 5
```

- Assume the branch is taken: How many cycles to execute this code?
- Assume the branch is fallthrough: How many cycles execute this code?

Calculating performance with control hazards

Assume the first branch is taken 50% of the time and the loop iterates 100 times, and forwarding for all data hazards.

1. How many cycles does the code take assuming detect and stall for control hazards?
2. How many cycles does the code take assuming speculate and squash where all branches are predicted fall through?
3. How many cycles does the code take assuming speculate and squash where backward branches are predicted taken and forward branches fall through?

```
add   1  2  3
beq   1  5  1
lw    6  4  1
add   3  4  5
beq   5  7  -5
```