TU WIEN Informatics

Systems Computing Embedded

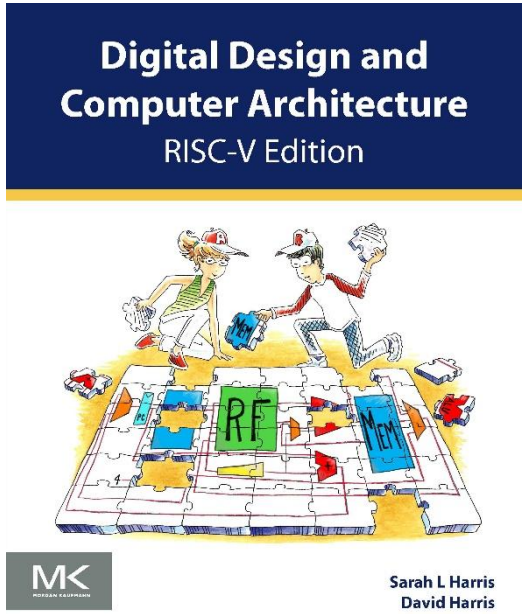
Computer Systems

Advanced Processor Pipelines 1

Daniel Mueller-Gritschneder

07.04.2025

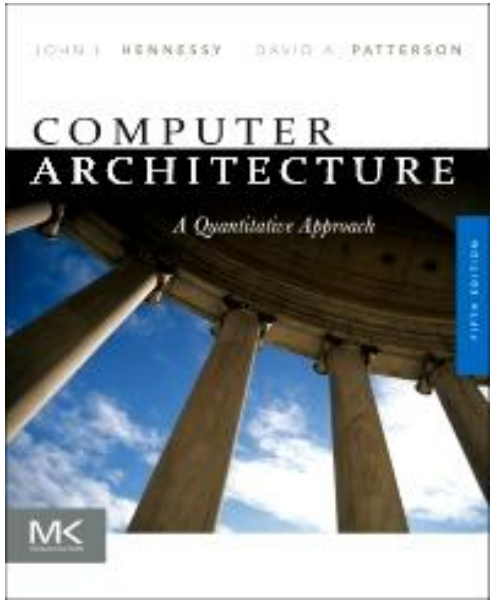
Sources



This book covers the basics of how to design a simple in-order scalar processor pipeline in detail in hardware.

- Literature: „Digital Design and Computer Architecture: RISC-V Edition“, by Sarah L. Harris and David Harris
  - https://shop.elsevier.com/books/digital-design-and-computer-architecture-risc-v-edition/harris/978-0-12-820064-3
  - https://pages.hmc.edu/harris/ddca/ddcarv.html (Includes resources for students!)
  - They also provide slideshows – the basis for ours! You can investigate extended version at their website.
- Available at TU’s library: https://catalogplus.tuwien.at/permalink/f/qknpf/UTW_alma2113990399000336

Sources



So-called application processors have many additional features:
Branch prediction, Out of order execute, Scoreboard, Superpipelining, Multi-issue, Superscalar, VLIW, Multi-threading, …

Disclaimer: The book provides advanced concepts from real complex processor designs. We only study the concepts at a high level. For simplicity, the used pipeline models in this lecture are reduced strongly in complexity.

But: We will have a look at some current RISC-V processor designs

Literature: „**Computer Architecture A Quantitative Approach”** 5th Edition - September 16, 2011
Authors: John L. Hennessy, David A. Patterson eBook ISBN: 9780123838735

- https://shop.elsevier.com/books/computer-architecture/hennessy/978-0-12-383872-8
- Available at TU’s library: https://catalogplus.tuwien.at/permalink/f/8agg25/TN_cdi_askewsholts_vlebooks_9780123838735

Content – Session 1

- Short Recap: RISC-V Assembly
- Five-Stage In-order Scalar Processor Pipeline
  - Pipelined Execution & Stages
  - Data Hazards & Forwarding Paths
  - Control Hazards
- Branch Prediction
  - Static Predictors: Taken / Not taken /BTFNT
  - Branch Target Buffer
  - Dynamic Predictors: 1 bit / 2 bit

- A look at a real RISC-V processor – CVA6
- A look at a real RISC-V processor – ESP32- C3
- Trap Handling

Optional, not relevant for exam

Short Recap: RISC-V Assembly

Writing a small assembly function: abs_value

- Example C-Code 1

```c
// Computes the absolute value
int abs_value(int a) {
    if (a<0)
        a=0-a;
    return a;
}
```

JALR rd,rs,imm
Behavior:
rd=PC+4
PC=rs+sign_extend(imm)

JALR x0,ra,0
PC=ra

- RISC-V Code
  - According to ABI a is given to the function in register a0
  - The function should also return a in register a0

```
abs_value:
        BGE a0,zero,abs_value_return # if a>=0
        SUB a0,zero,a0               # a=0-a
abs_value_return:
        RET                          # JR ra
```

function return which is a pseudo instruction for
JR ra
which is a pseudo instruction for
JALR x0,ra,0

Writing a small assembly function: vec_add

- Example C-Code 3

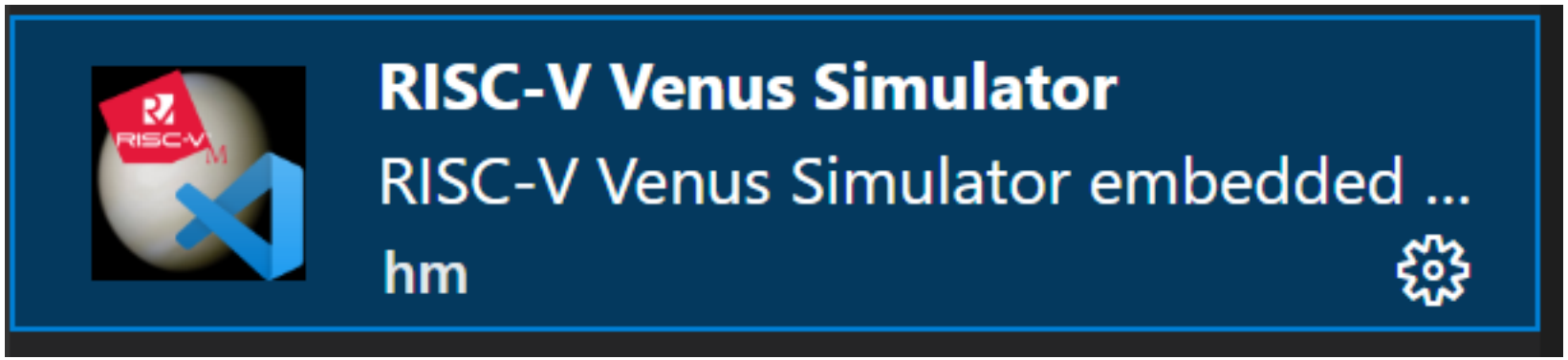
```c
// vector addition of 4-element integer vectors
void vec_add(int[4] a, int[4] b, int[4] c) {
  unsigned int i;
  for (i=0;i<4;i++) {
    c[i] = a[i] + b[i];
  }
}
```

RISC-V Code

```
# base address of a: a0,
# base address of b: a1,
# base address of c: a2,
# i: t0,  constant 4: t3
vec_add:
  LI t0,0        # i=0
  LI t3,4        # t3=4
vec_add_for:
  LW t1,0(a0)     # t1 = a[i]
  LW t2,0(a1)     # t2 = b[i]
  ADD t1,t1,t2    # t1 = a[i] + b[i]
  SW t1,0(a2)     # c[i] = t1
  ADDI a0,a0,4    #next element is base address + 4
  ADDI a1,a1,4    #next element is base address + 4
  ADDI a2,a2,4    #next element is base address + 4
  ADDI t0,t0,1    # i++
  BLTU t0,t3,vec_add_for # for (i < 4)
  RET    # void return
```

RISC-V Simulator

- Visual Studio Code



Extensions -> Venus Simulator for RISC-V Assembly

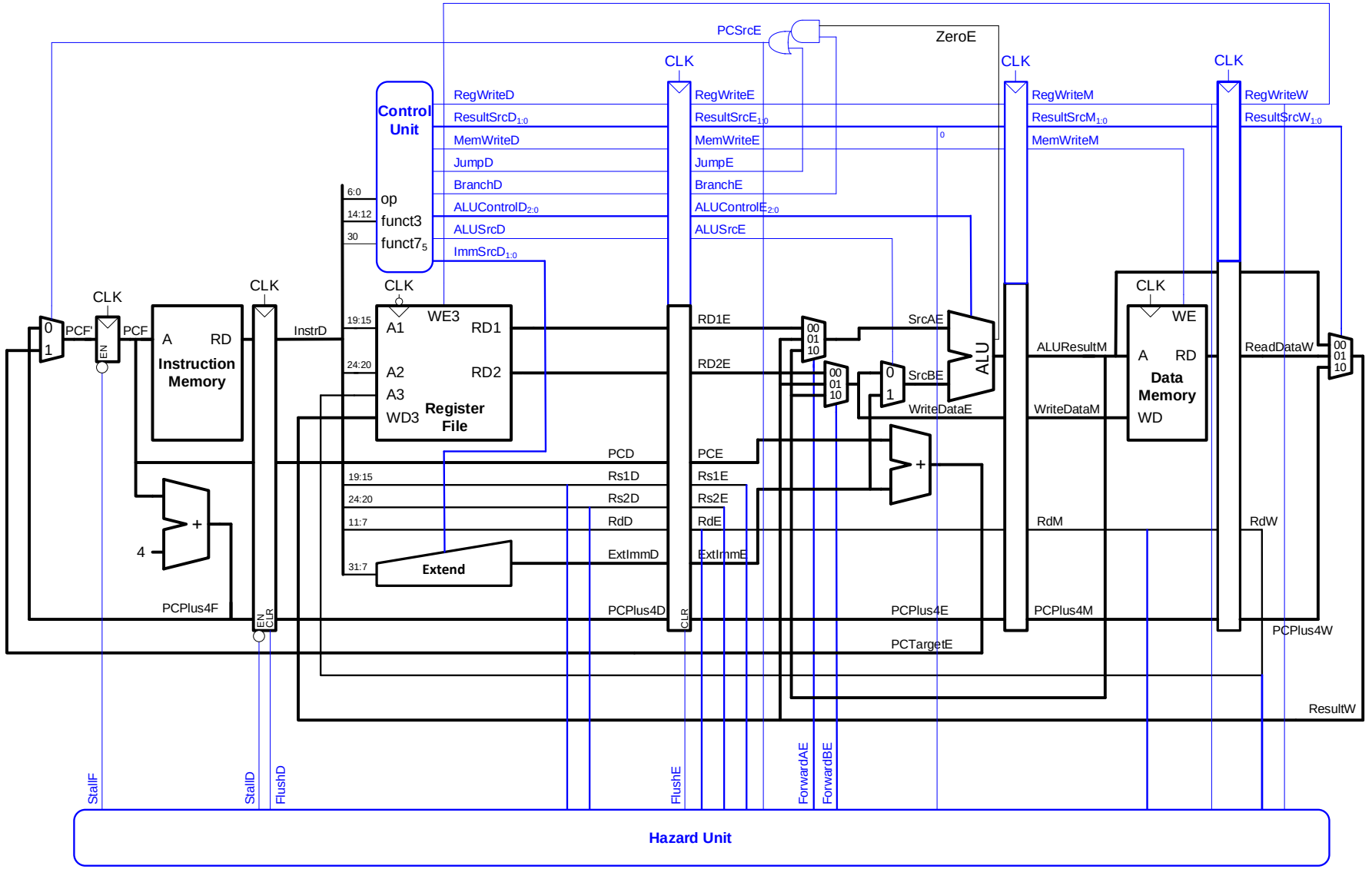
Five-Stage Scalar In-order Processor Pipeline

Pipelined execution

- We break down instructions in sub-computations and place them into stages (s)

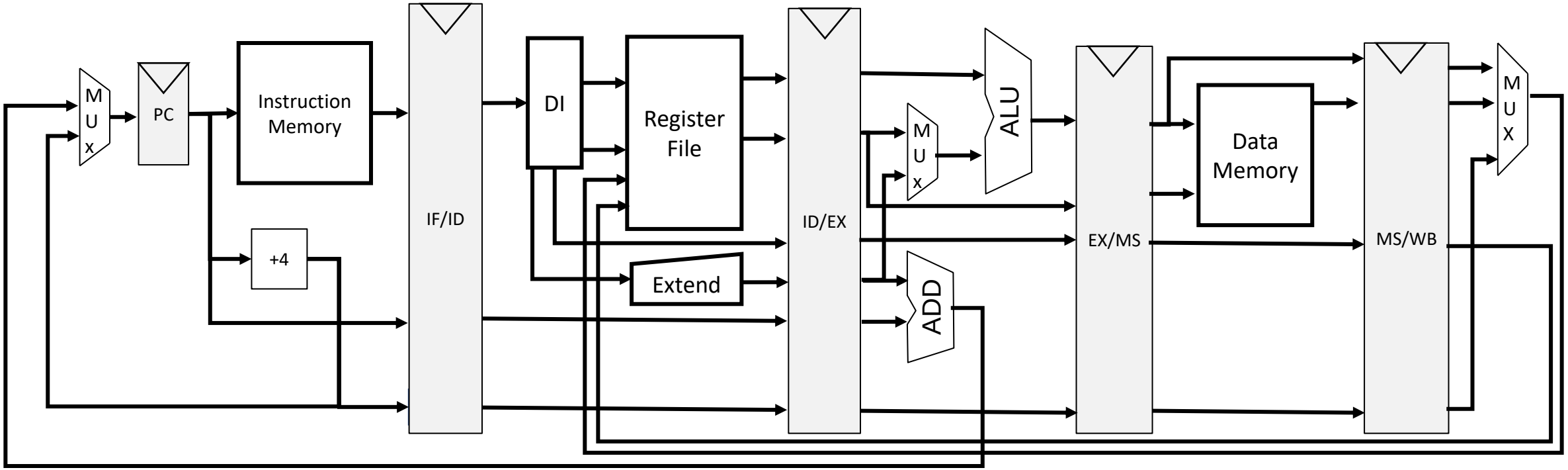
- We execute the instructions in a pipelined fashion („Fließband“)

Recap: Five-Stage In-order Scalar Processor Pipeline (Harris & Harris)



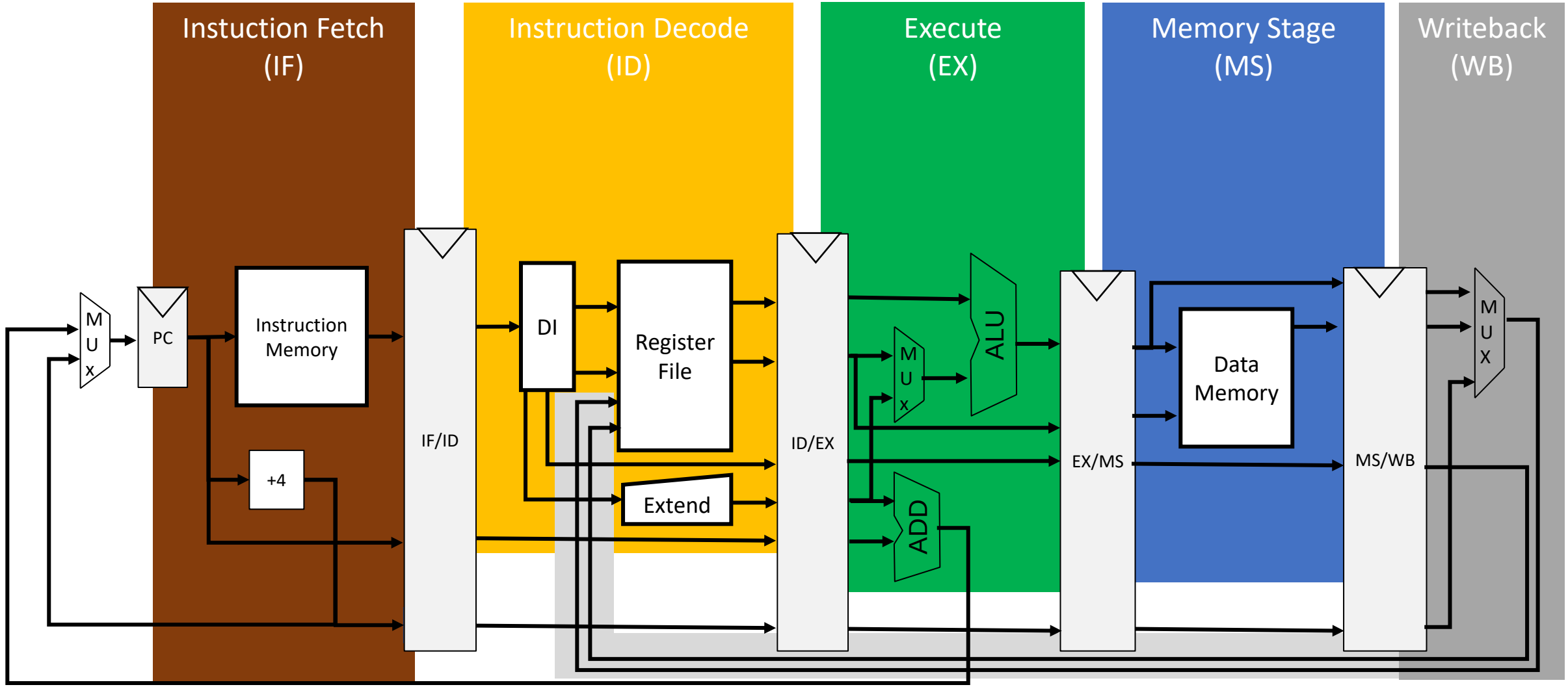
Five-stage Pipeline - Data Signal Busses

- Data path scheme of the pipeline:
  - We omit all control signals.
  - We are only interested how instructions can „flow“ through the pipeline (data signal busses)



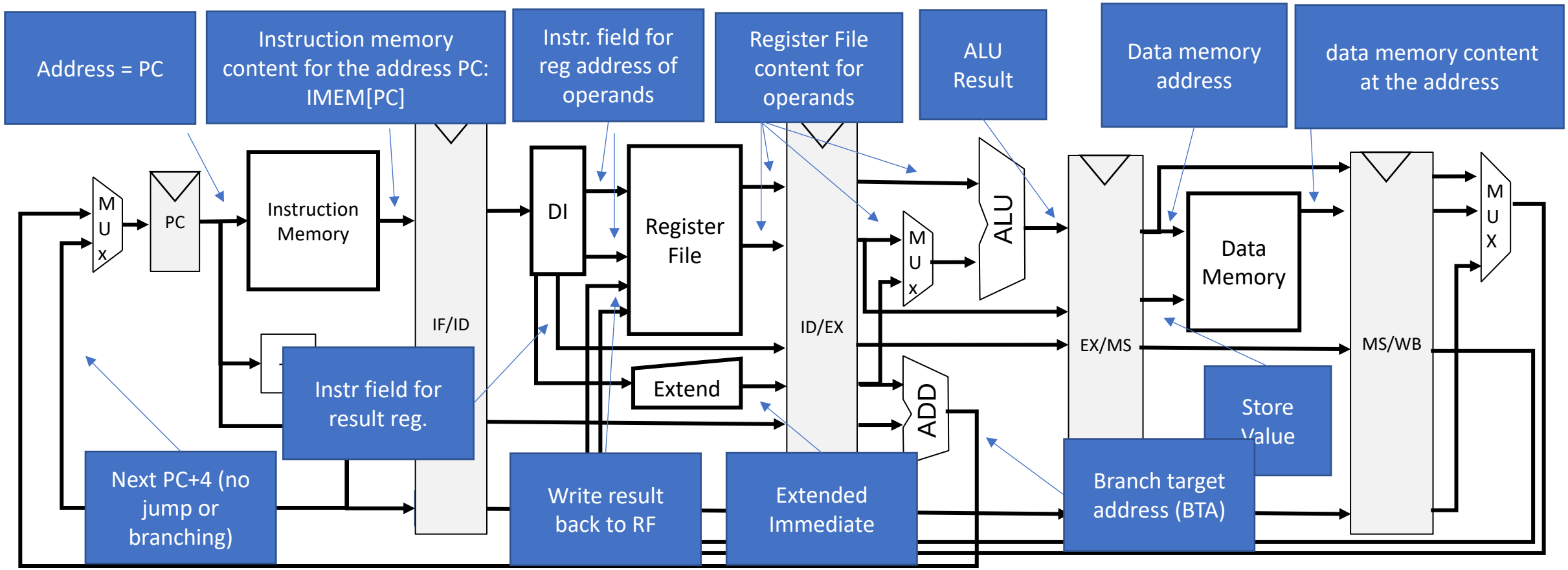
Five-stage Pipeline - Stages

- Stages:



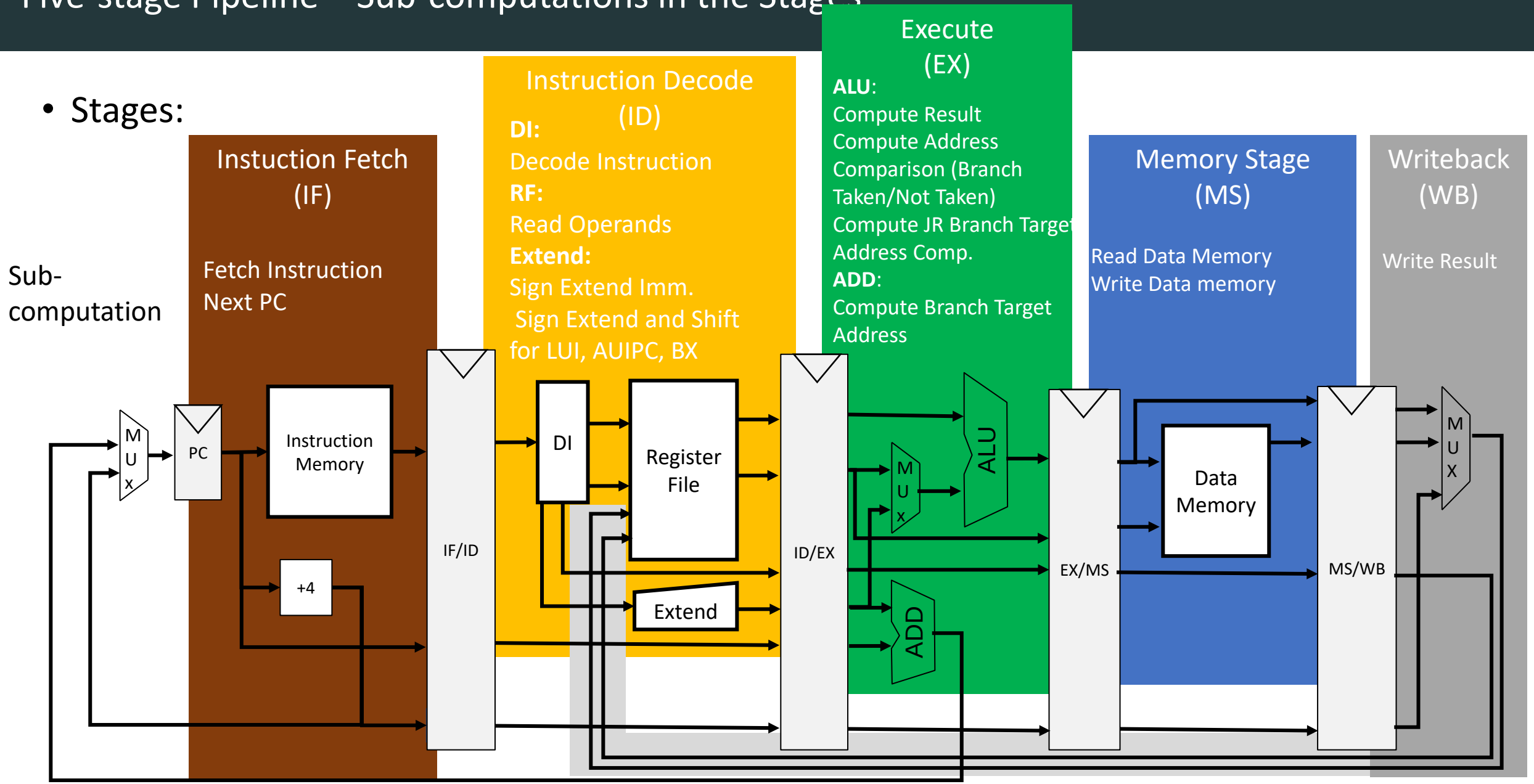
Five-stage Pipeline - Data Signal Busses

- Data path scheme of the pipeline:
  - We omit all control signals.
  - We are only interested how instructions can „flow“ through the pipeline (data signal busses)



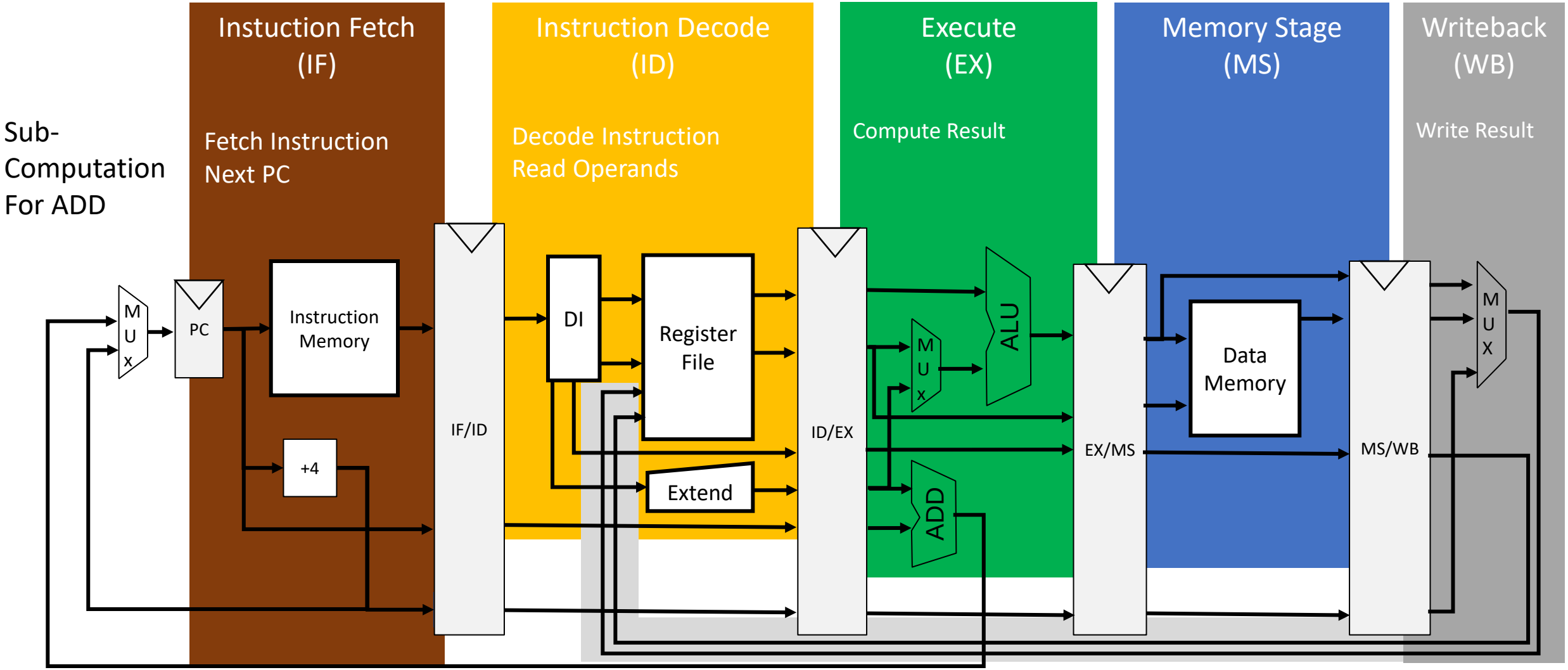
Five-stage Pipeline – Sub-computations in the Stages

- Stages:



Five-stage Pipeline – Sub-computations in the Stages (Example ADD)

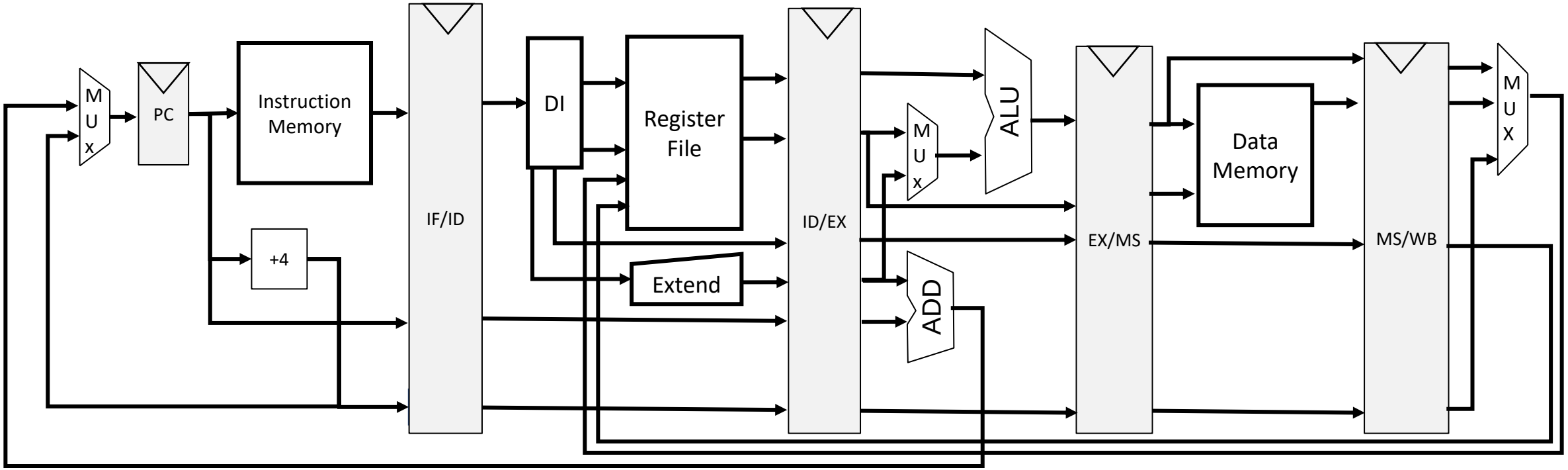
- Instructions do not require all subcomputations, e.g. ADD



Five-stage Pipeline – Example Program

- Example program

```
#int test1(int *x, int i) {return x[i]+i;}
test1:
  SLLI a2,a1,2  # a2=i*4
  ADD a2,a0,a2   # baseaddr+offset i*4
  LW a0,0(a2)  # a0 = x[i]
  ADD a0,a0,a1   # a0= x[i] + i
  RET
```

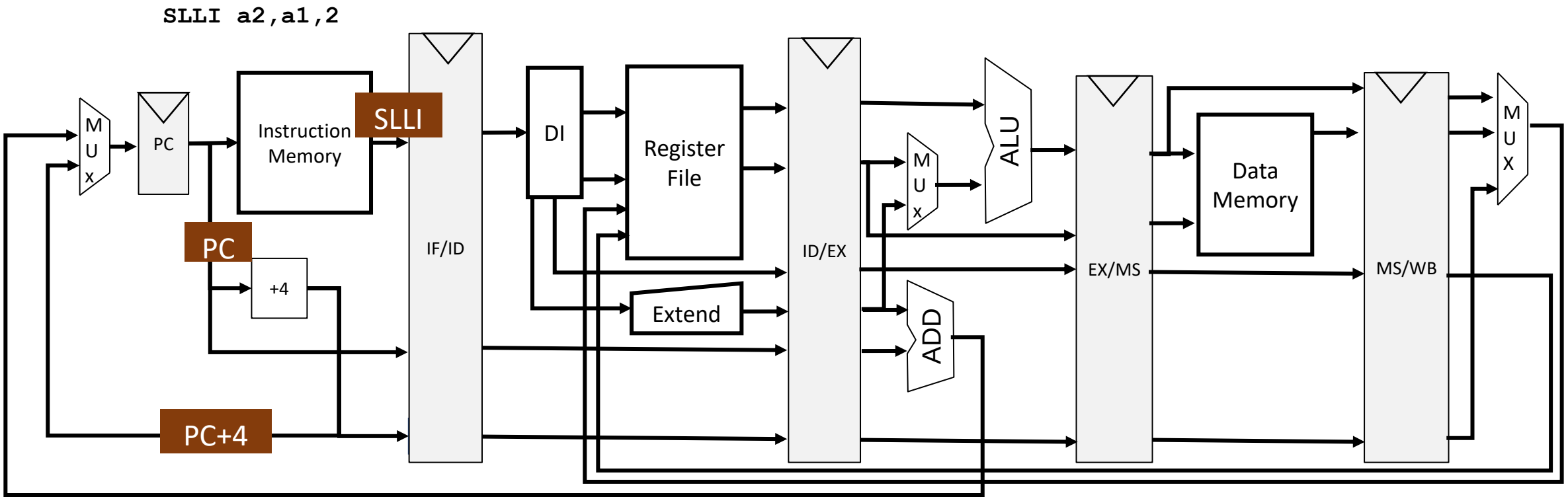


Five-stage Pipeline – Example Program – Cycle 1

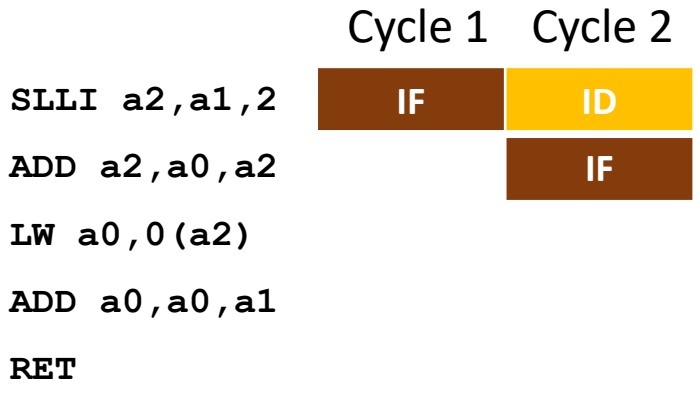
Cycle 1

```
SLLI a2,a1,2
ADD a2,a0,a2
LW a0,0(a2)
ADD a0,a0,a1
RET
```

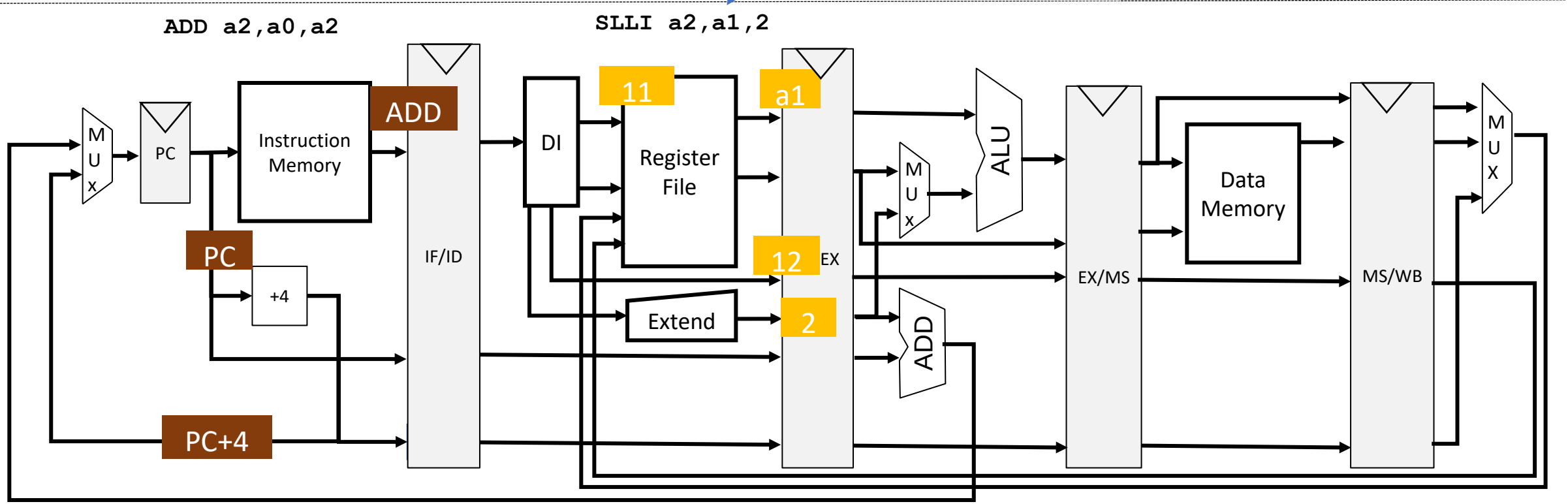
IF



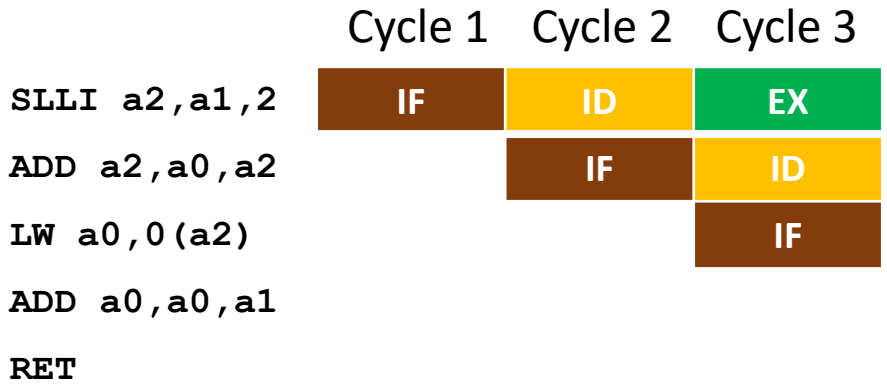
Five-stage Pipeline – Example Program – Cycle 2



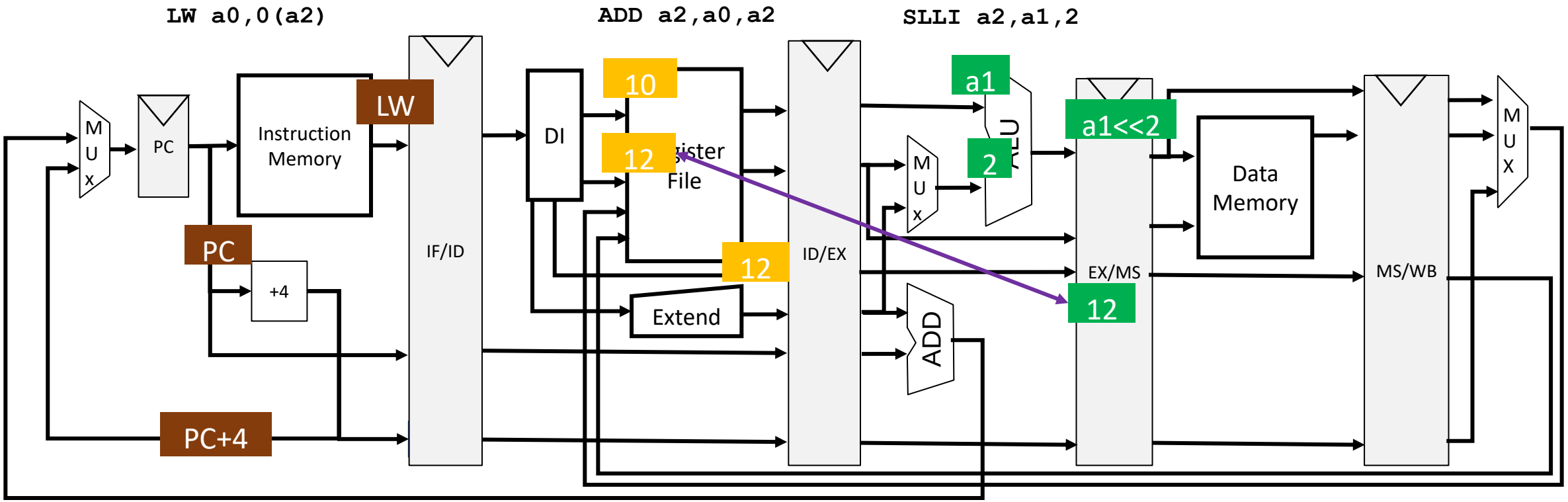
This is x11



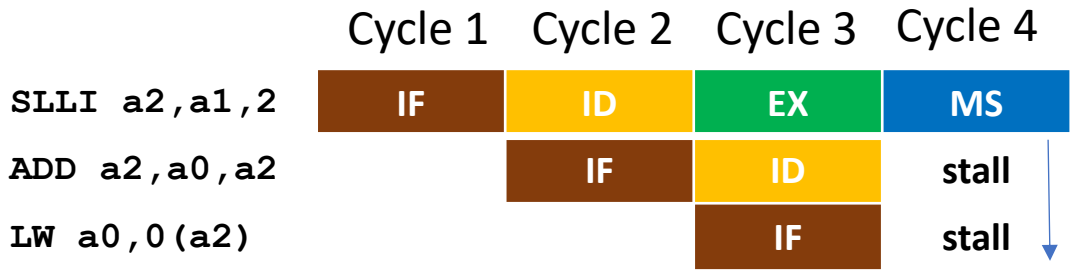
Five-stage Pipeline – Example Program – Cycle 3



Data hazard: a2 not yet updated by SLLI -> Stall ADD because it cannot leave ID stage



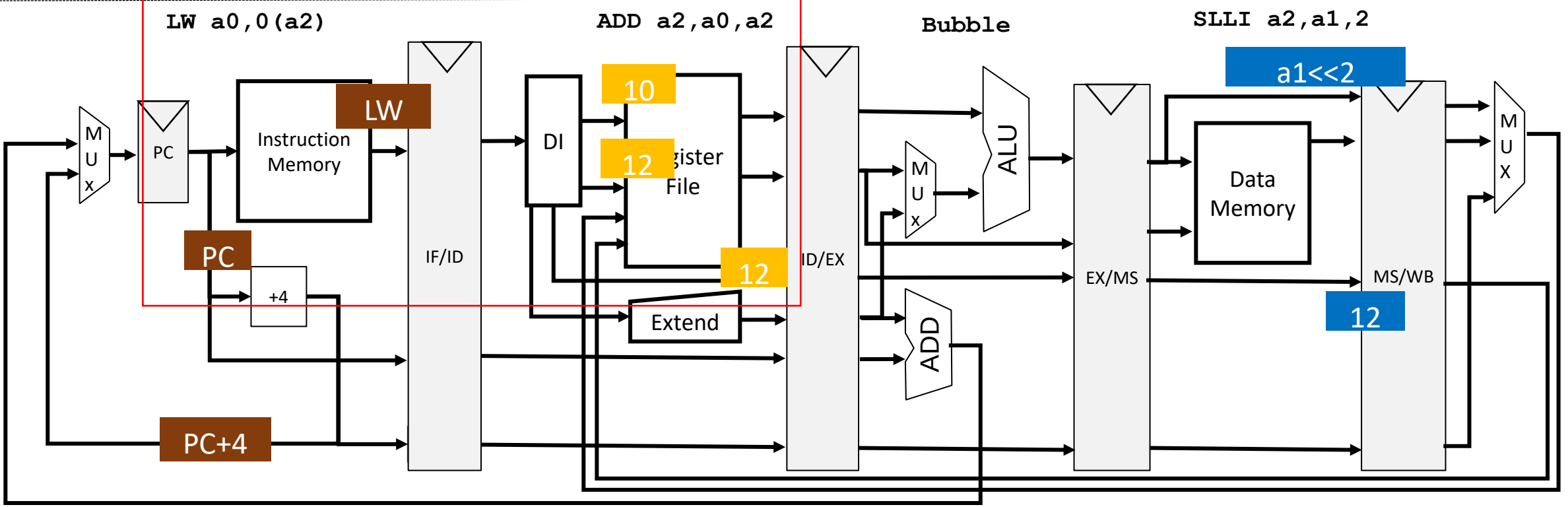
Five-stage Pipeline – Example Program – Cycle 4



Stalls backpropagate in the pipeline to following instructions

There is no instruction in the execute stage -> Insert a so-called Bubble (NOP)

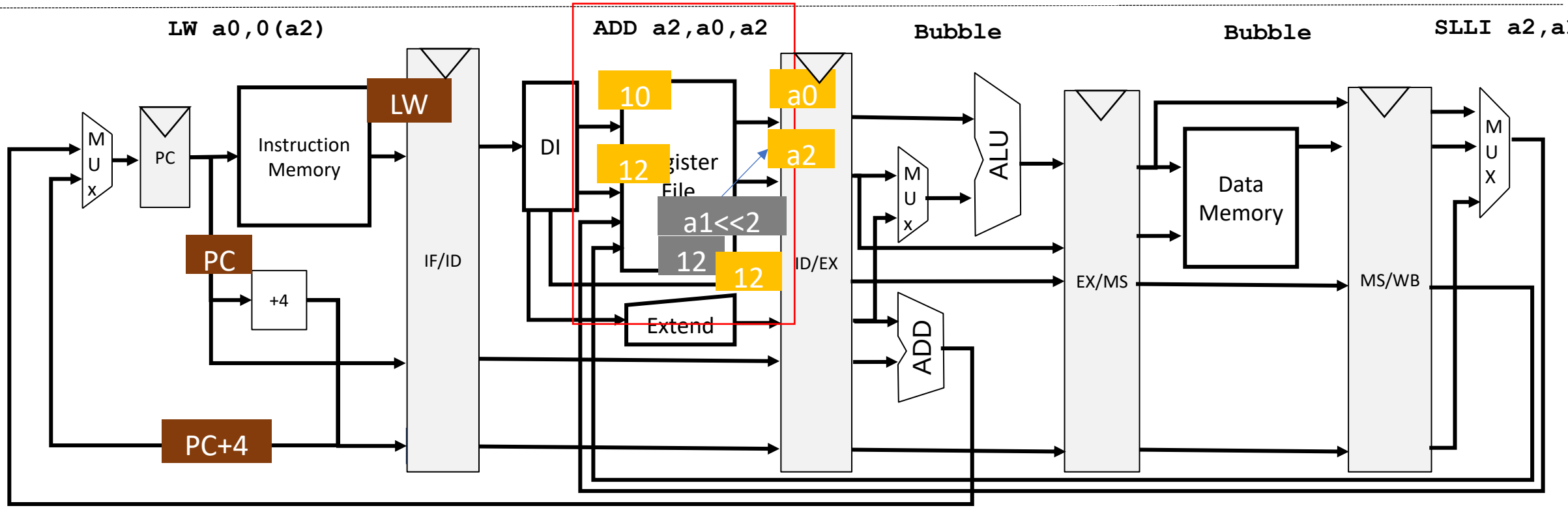
ADD and LW stall



Five-stage Pipeline – Example Program – Cycle 5

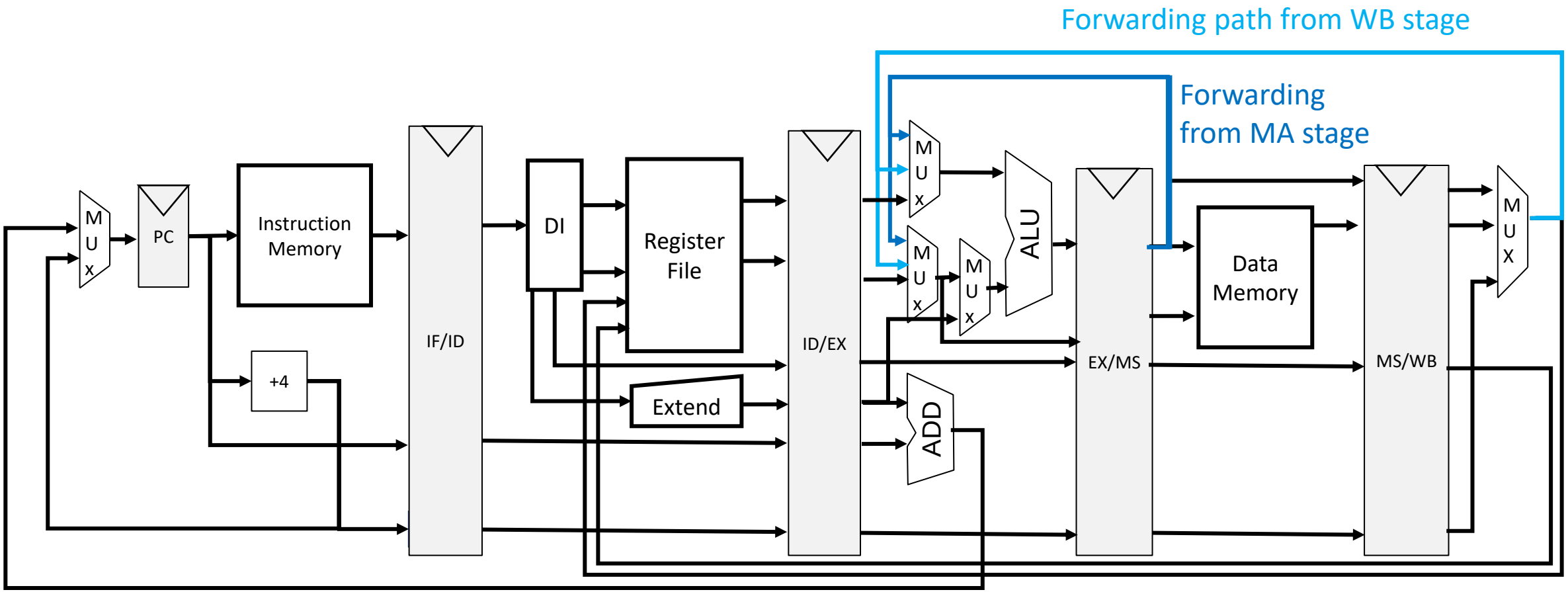
	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
`SLLI a2,a1,2`	IF	ID	EX	MS	WB
`ADD a2,a0,a2`		IF	ID	stall	stall
`LW a0,0(a2)`			IF	stall	stall
`ADD a0,a0,a1`					
`RET`					

ADD can complete ID stage -> stop stalling

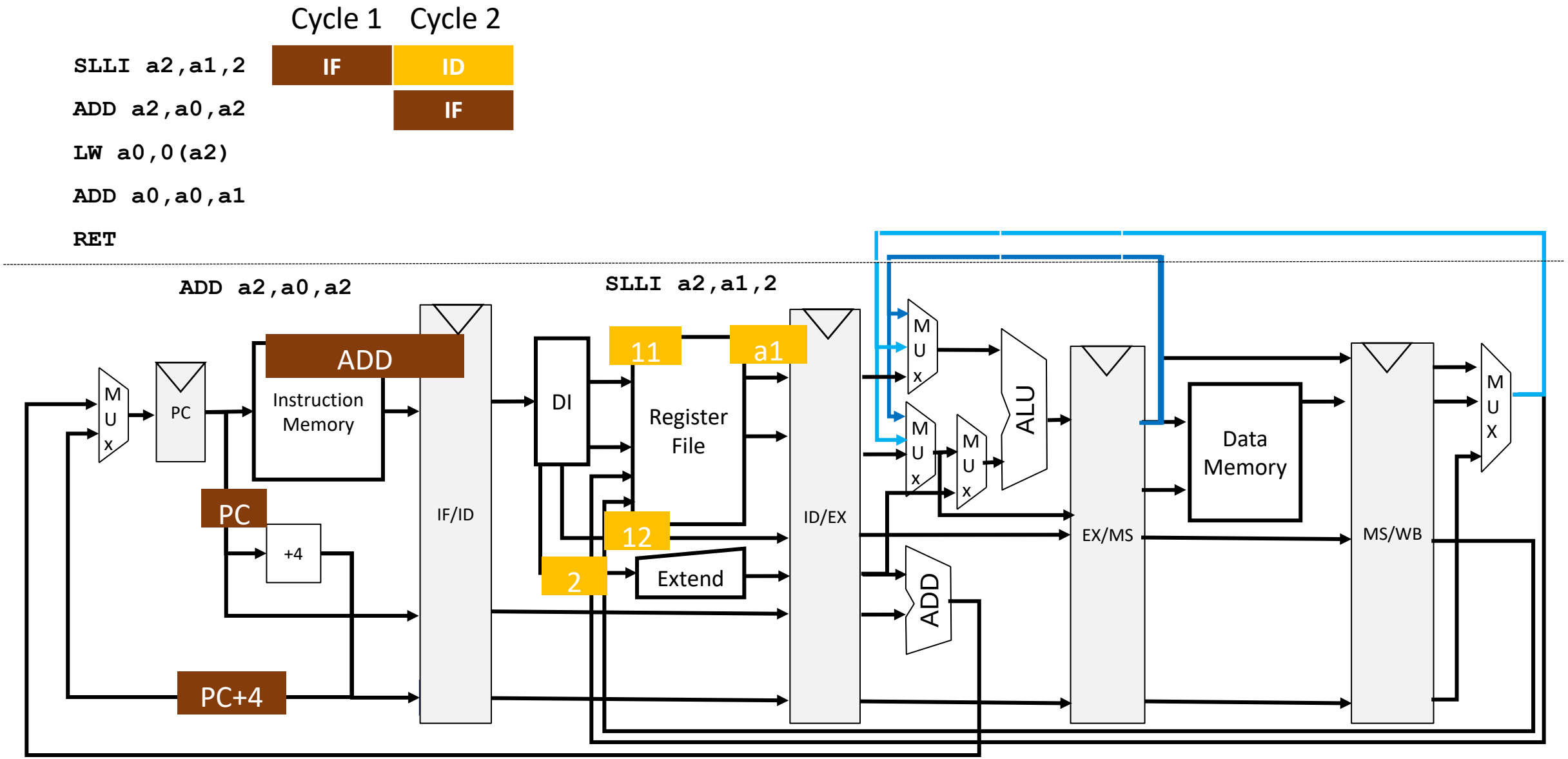


Five-stage Pipeline with Forwarding Path

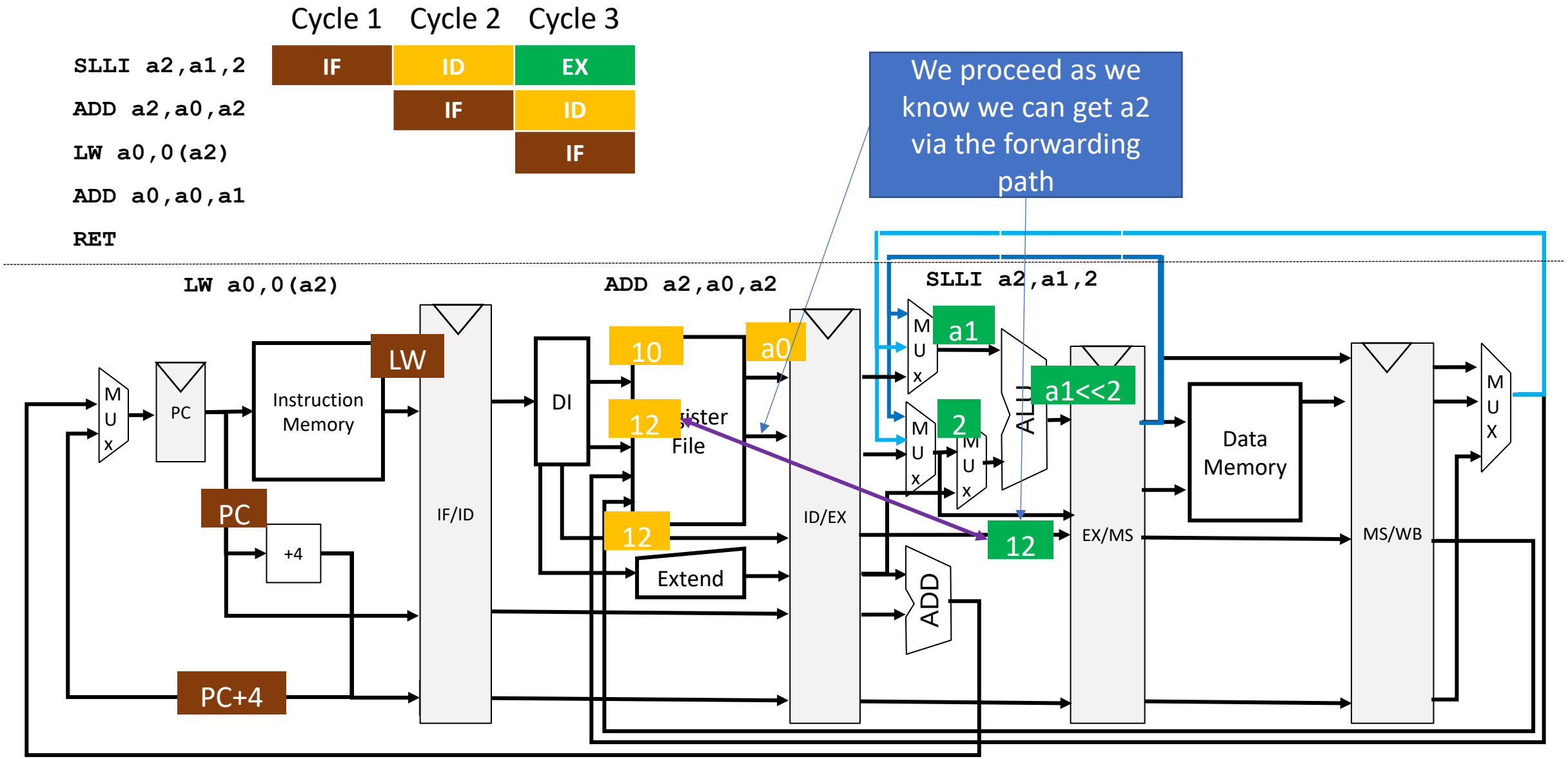
- Data hazards can be effectively mitigated using a forwarding path
- While named „forwarding path“ the signal buses go „back“ in the pipeline



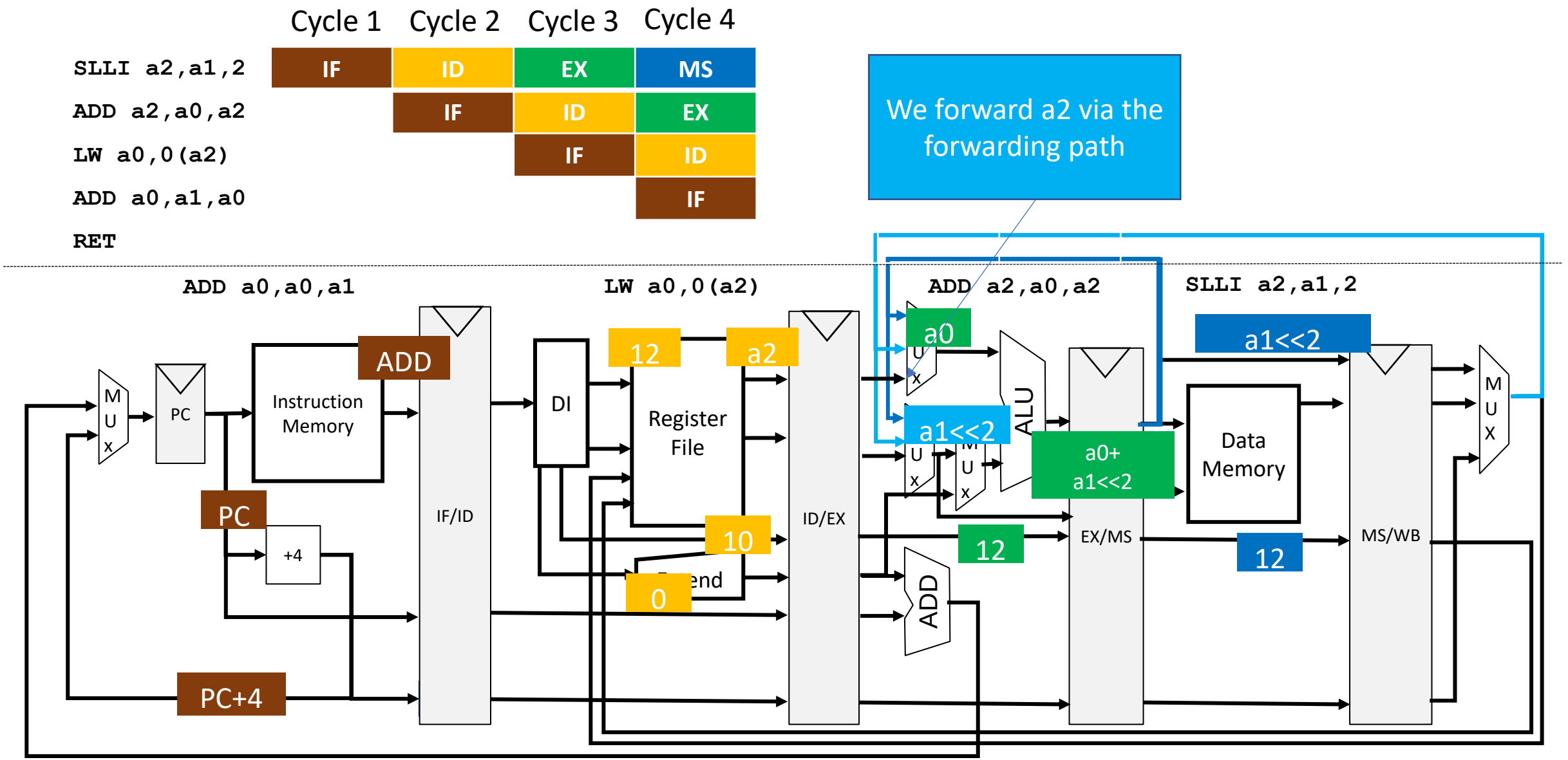
Five-stage Pipeline with Forwarding Path – Example Program – Cycle 2



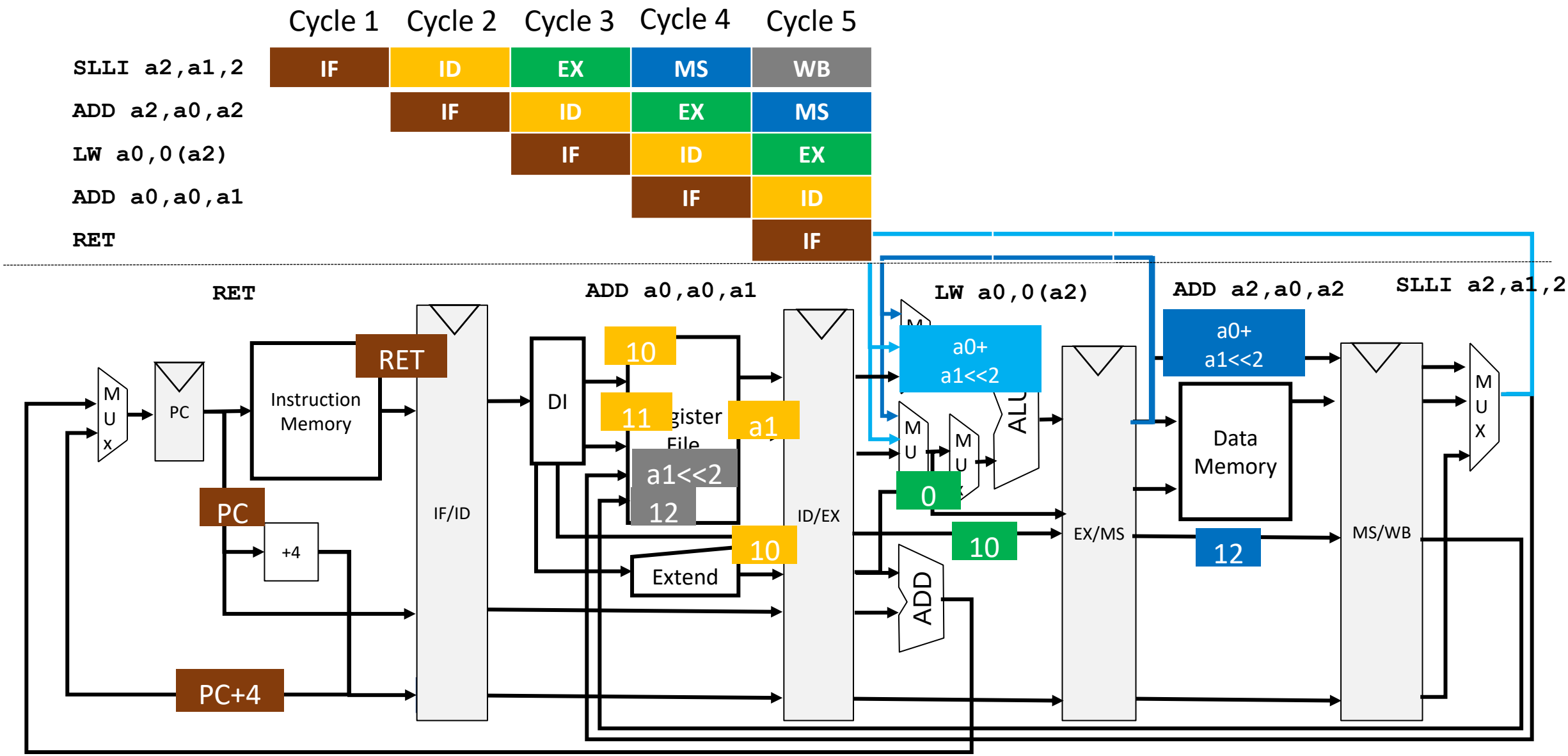
Five-stage Pipeline with Forwarding Path – Example Program – Cycle 3



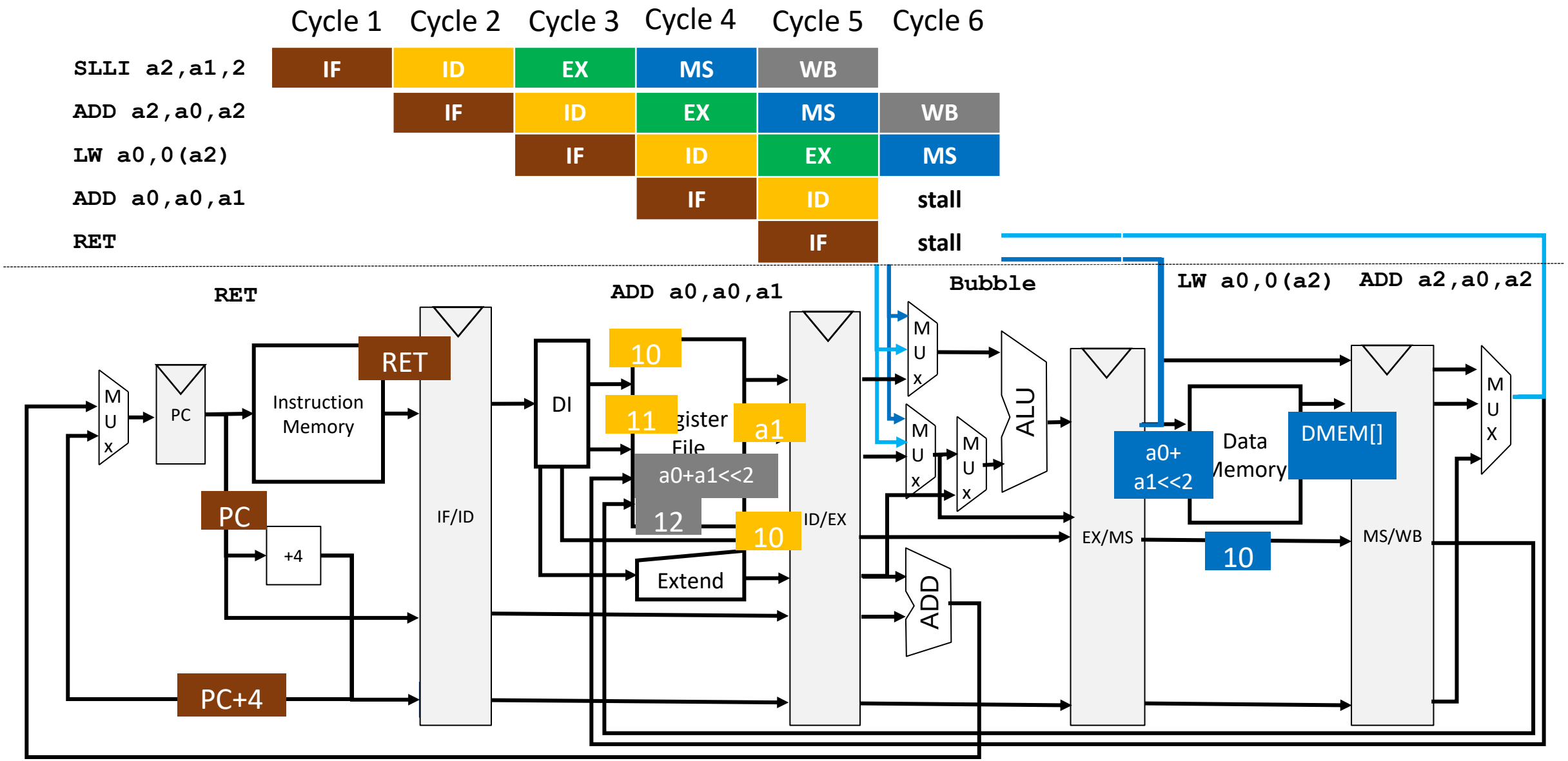
Five-stage Pipeline with Forwarding Path – Example Program – Cycle 4



Five-stage Pipeline with Forwarding Path – Example Program – Cycle 5

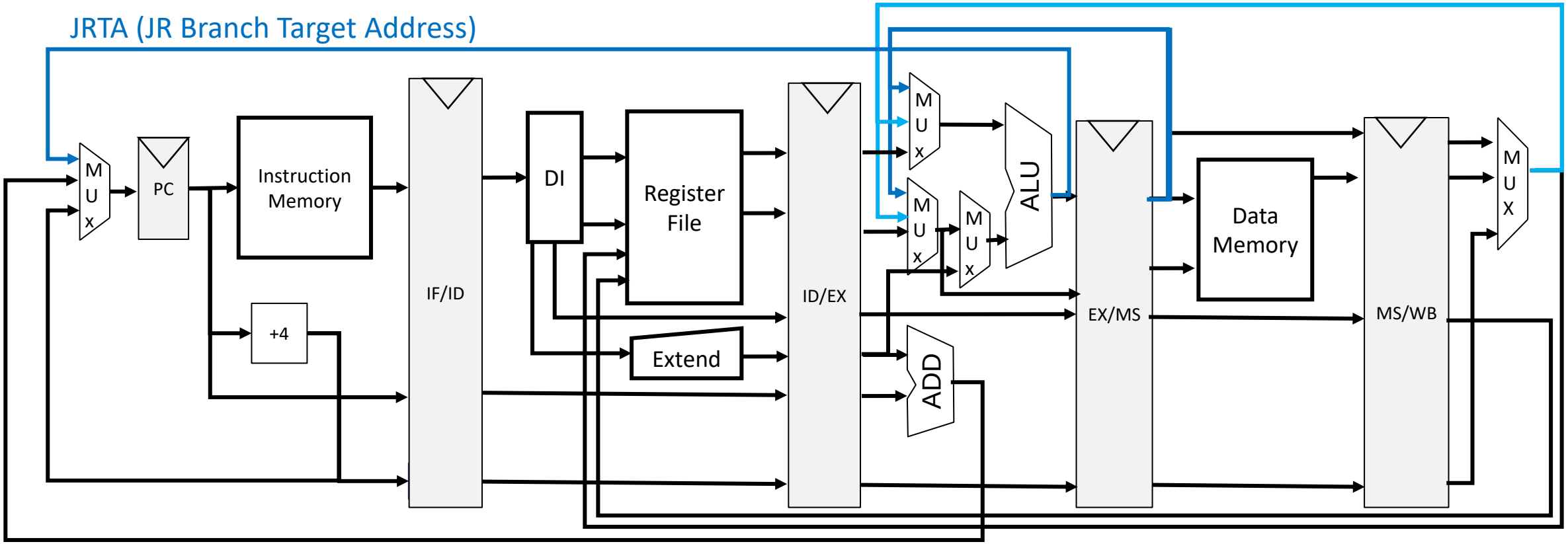


Five-stage Pipeline with Forwarding Path – Example Program – Cycle 6

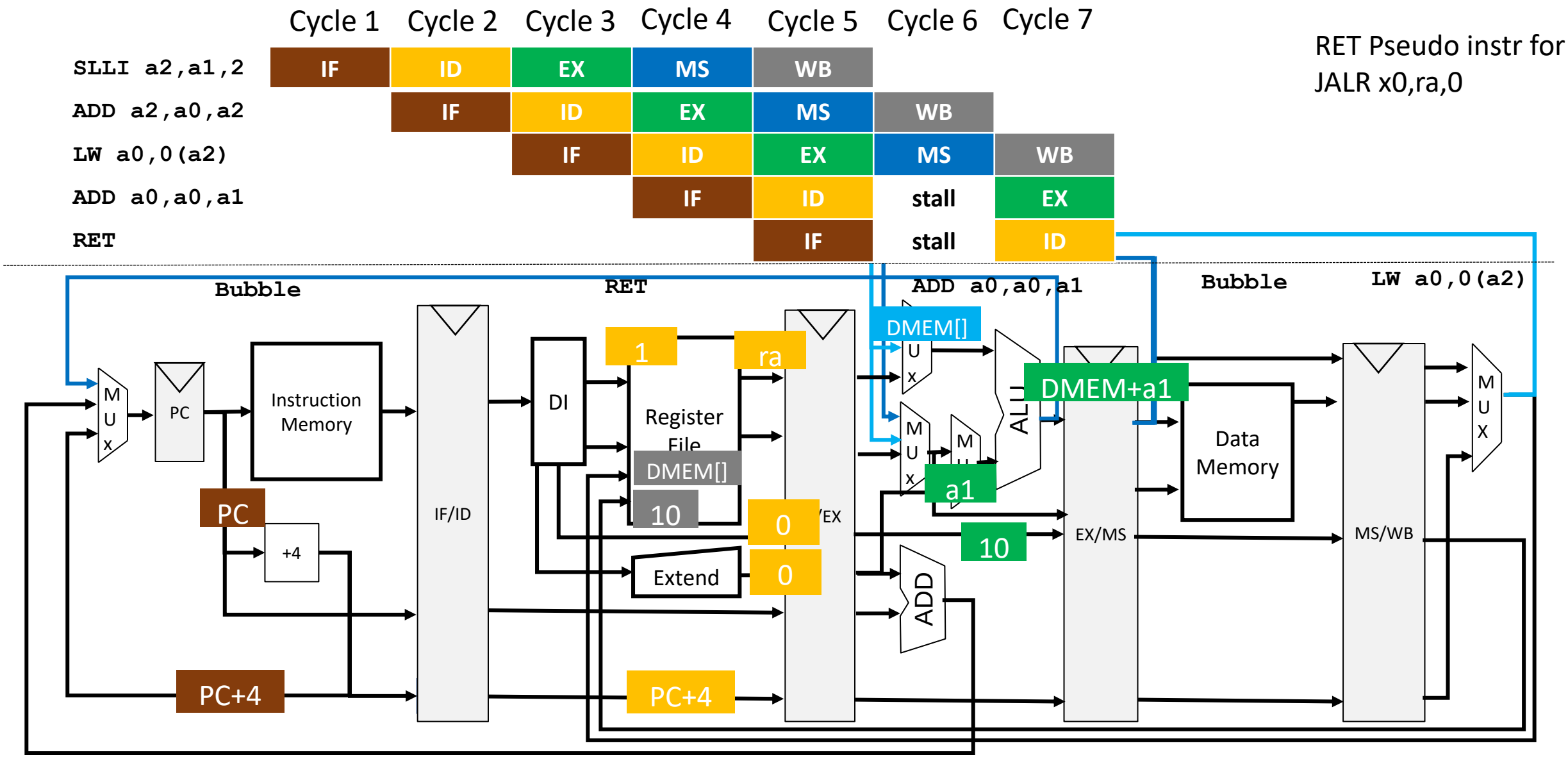


Five-stage Pipeline with Forwarding Path and JR

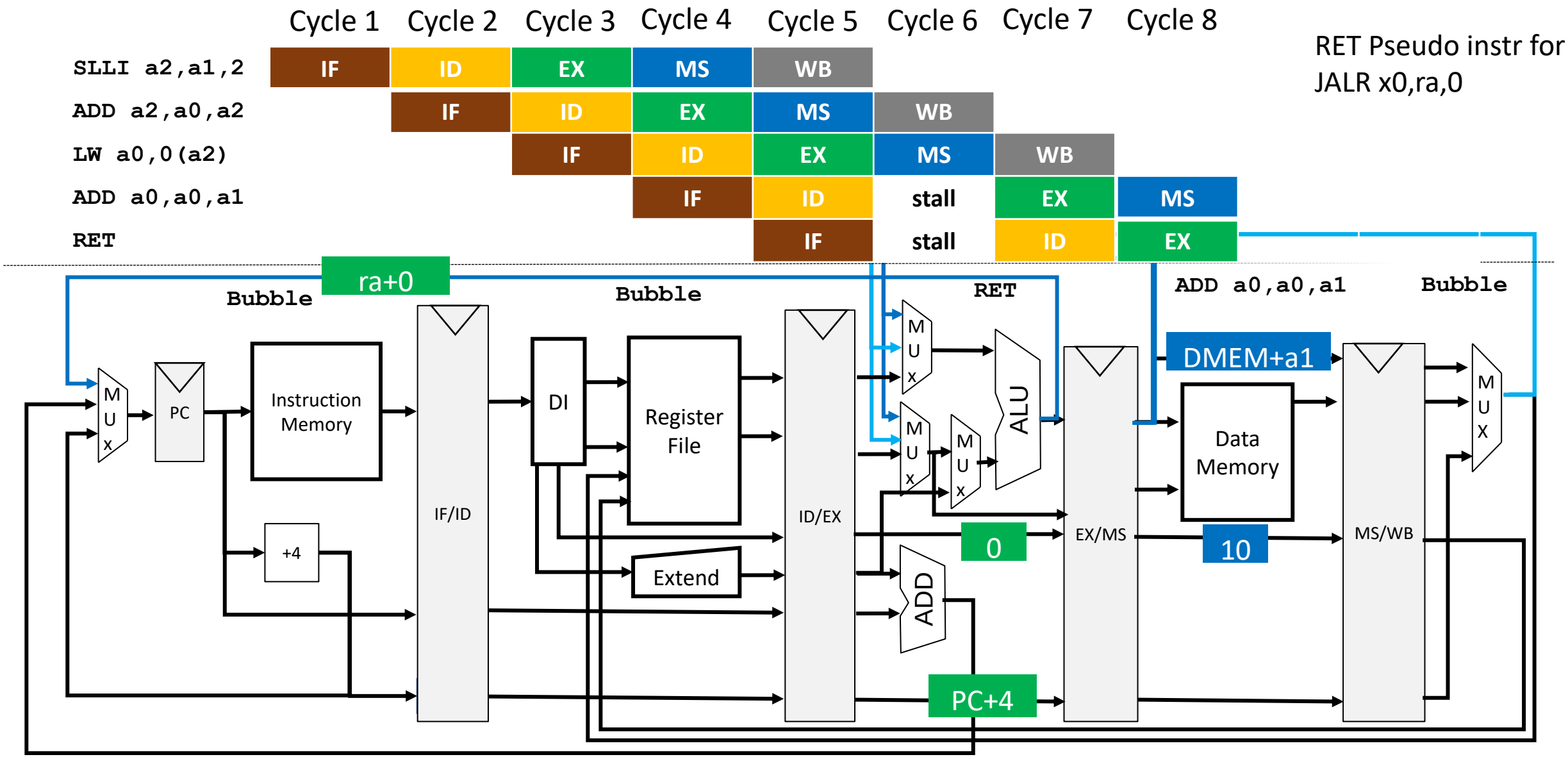
- RET is a pseudo-instruction for jump register JR ra, which is a pseudo instruction for JALR x0,ra,0
- The Harris pipeline does not support to load a register value into PC
- We need another bus for implementing the JR instruction



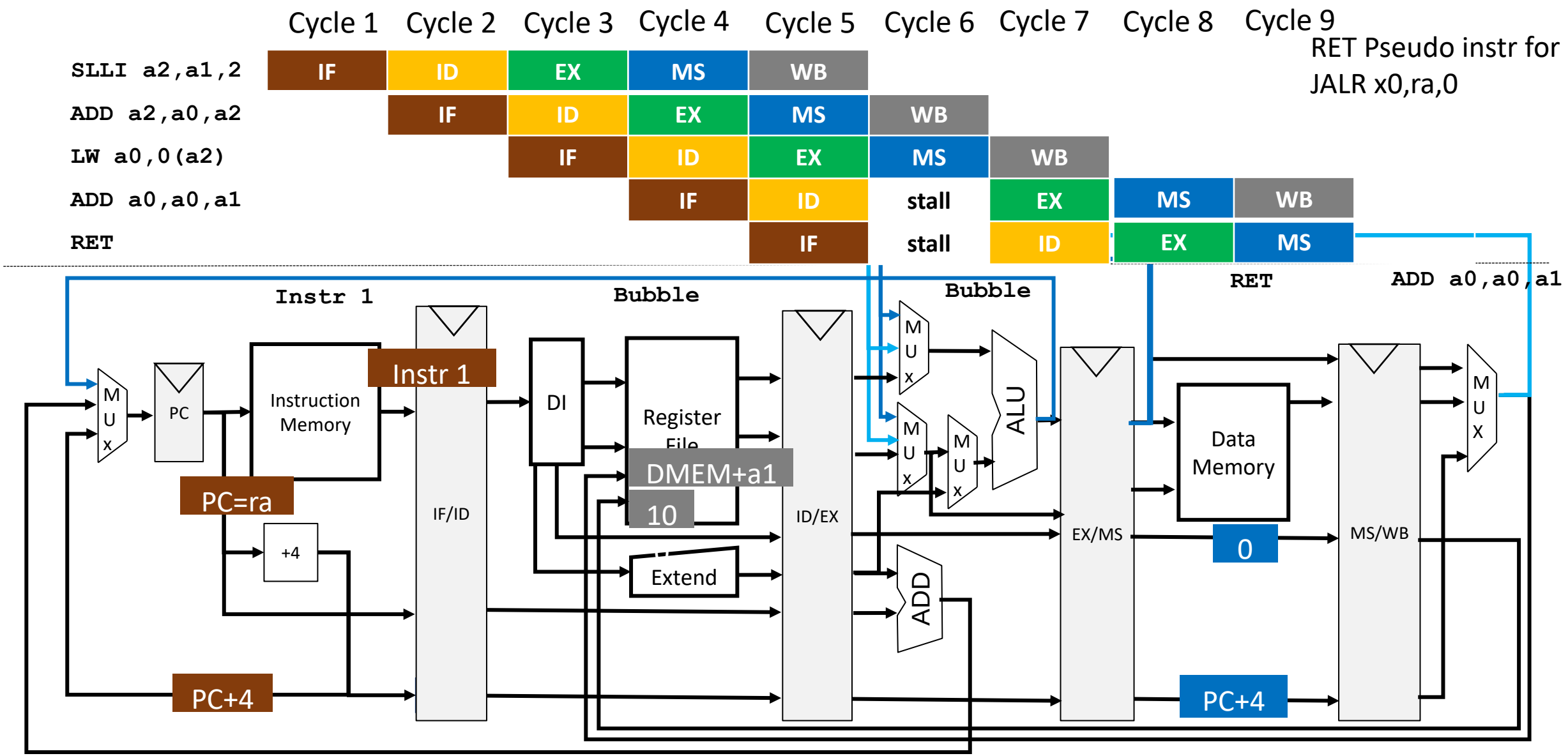
Five-stage Pipeline with Forwarding Path – Example Program – Cycle 7



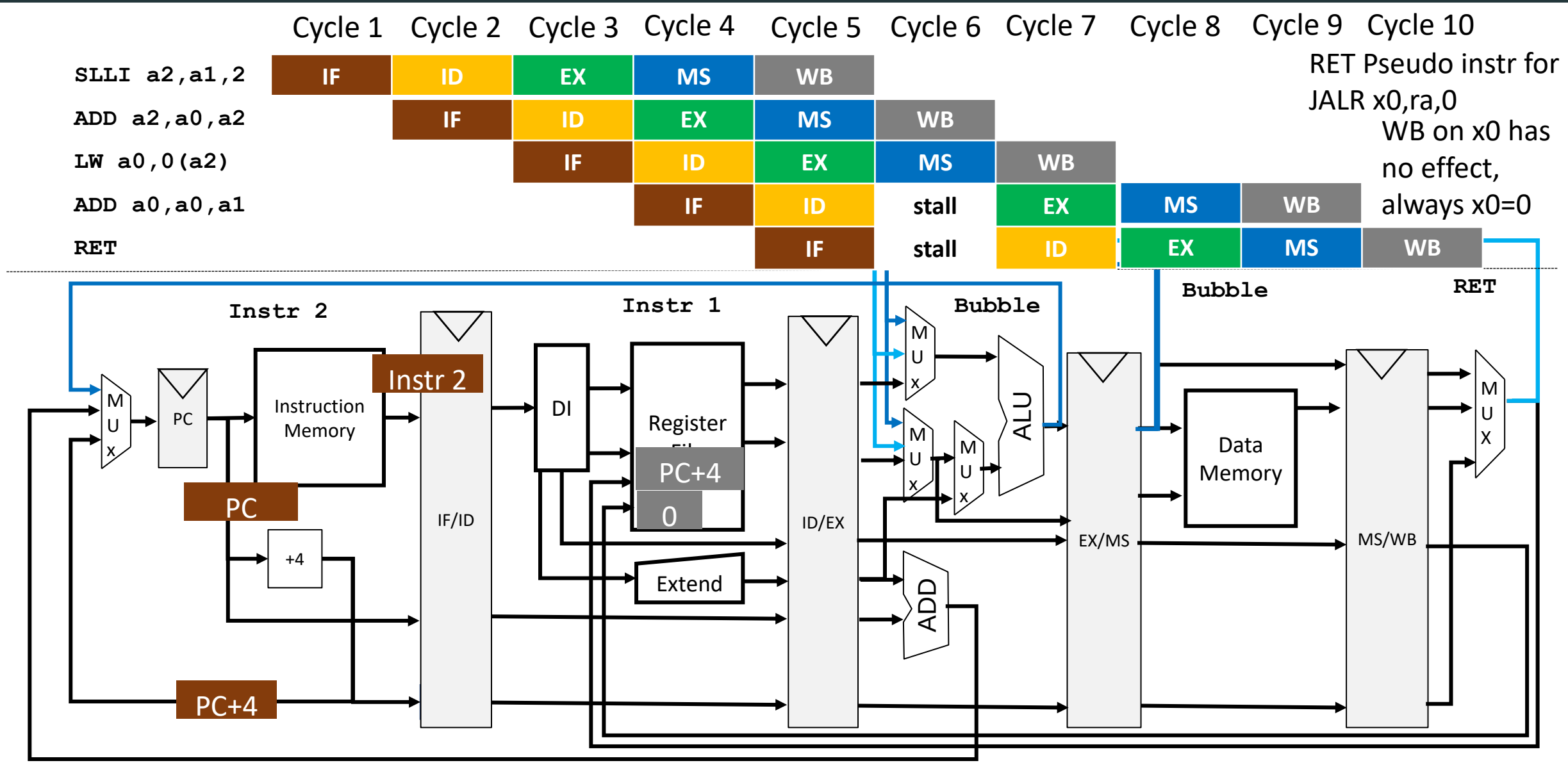
Five-stage Pipeline with Forwarding Path and JR - Example Program – Cycle 8



Five-stage Pipeline with Forwarding Path and JR - Example Program – Cycle 9

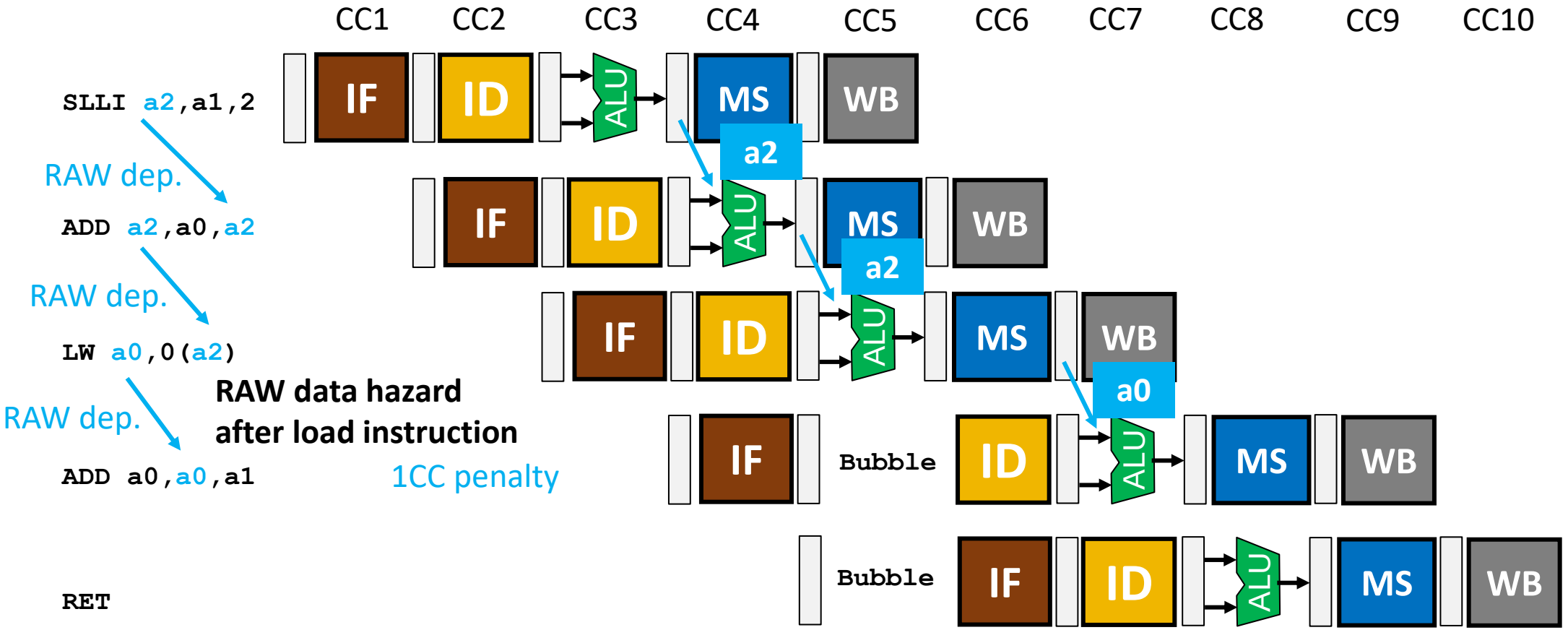


Five-stage Pipeline with Forwarding Path and JR - Example Program – Cycle 10



Five-stage Pipeline with Forwarding Path and JR - Pipeline Execution Diagram

- With forwarding path: Possible data hazard after load with penalty of 1 clock cycle (cc)



Read After Write (RAW) dependency or **„true dependency“**:
One instructions reads operand that is written as result of previous instructions.
Data hazard prevents the next instruction in the instruction stream from executing during its designated clock cycle.

Compiler Instruction Scheduling to Avoid RAW Data Hazards after Load Instructions

- Compiler often can move instructions to avoid RAW data hazards after loads
- Program order must not change (See next session)

➢Rarely data hazard penalty observed in five-stage pipeline with forwarding paths

➢Example:

```
vec_add_for:
  LW t1,0(a0)      # t1 = a[i]
  LW t2,0(a1)      # t2 = b[i]
   RAW        1CC penalty
  ADD t1,t1,t2     # t1 = a[i] + b[i]
  SW t1,0(a2)      # c[i] = t1
  ADDI a0,a0,4     #base address + 4
  ADDI a1,a1,4     #base address + 4
  ADDI a2,a2,4     #base address + 4
  ADDI t0,t0,1     # i++
   (…)
```

```
vec_add_for:
  LW t1,0(a0)      # t1 = a[i]
  LW t2,0(a1)      # t2 = b[i]
  ADDI t0,t0,1     # i++
  ADD t1,t1,t2     # t1 = a[i] + b[i]
  SW t1,0(a2)      # c[i] = t1
  ADDI a0,a0,4     #base address + 4
  ADDI a1,a1,4     #base address + 4
  ADDI a2,a2,4     #base address + 4
    (…)
```

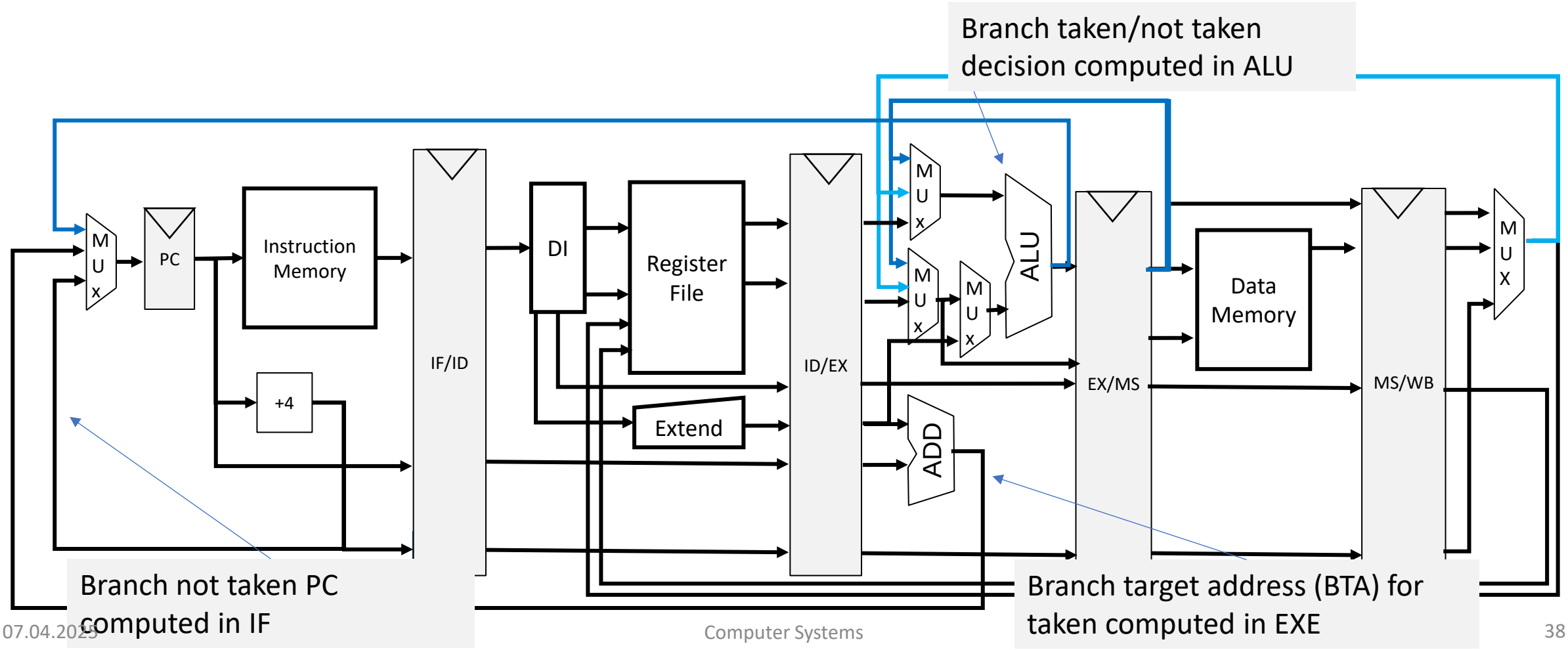
Control Hazard

Control Hazards

- **Control hazards** arise from instructions that change the PC
- **When the flow of instruction addresses is not sequential**
  - Unconditional branches (`jal, jalr`)
  - Conditional branches (`beq, bne, ...`)
  - Exceptions
- Possible approaches
  - **Stall** (impacts CPI)
  - **Move decision point** as early in the pipeline as possible (Extra HW)
  - **Predict** and hope for the best!
  - ***Delay decision*** *(requires compiler support)*
- Control hazards occur **less frequently** than data hazards,
  but there is **nothing as effective** against control hazards as forwarding is for data hazards

Control Hazards – Conditional Branches

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome (Branch taken/Not taken)
  - Next PC is either PC+4 (branch not taken) or PC+imm<<1 (branch taken)



Handling Control Hazards: Stall on Branch

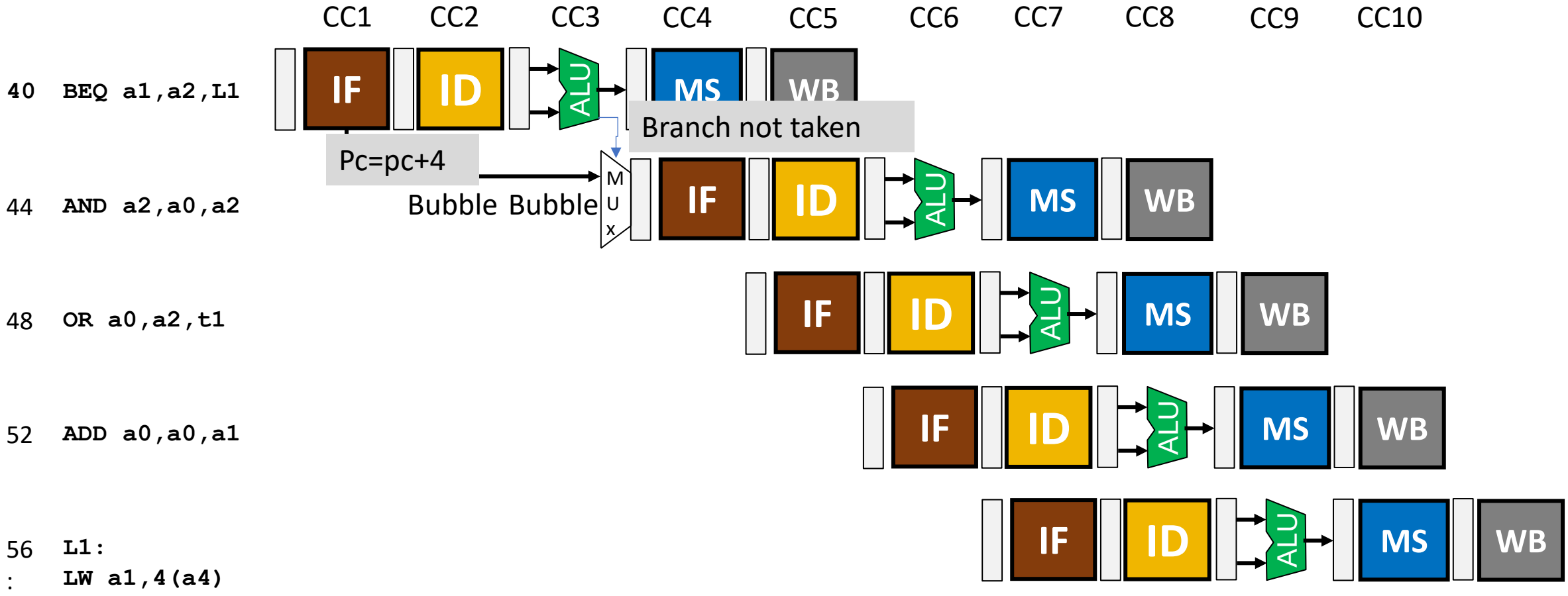
- **Conservative Approach:** **Wait** until branch outcome determined before fetching next instruction

> **Conservative approach**: Stall immediately after fetching a branch, wait until outcome of branch is known and fetch branch address.

- **Reducing Branch Delay:**
  - **E.g. Move Branch Decision to ID Stage:** **Extra hardware** so that we can test registers, calculate the branch address, and update the PC *during the second stage of the pipeline*

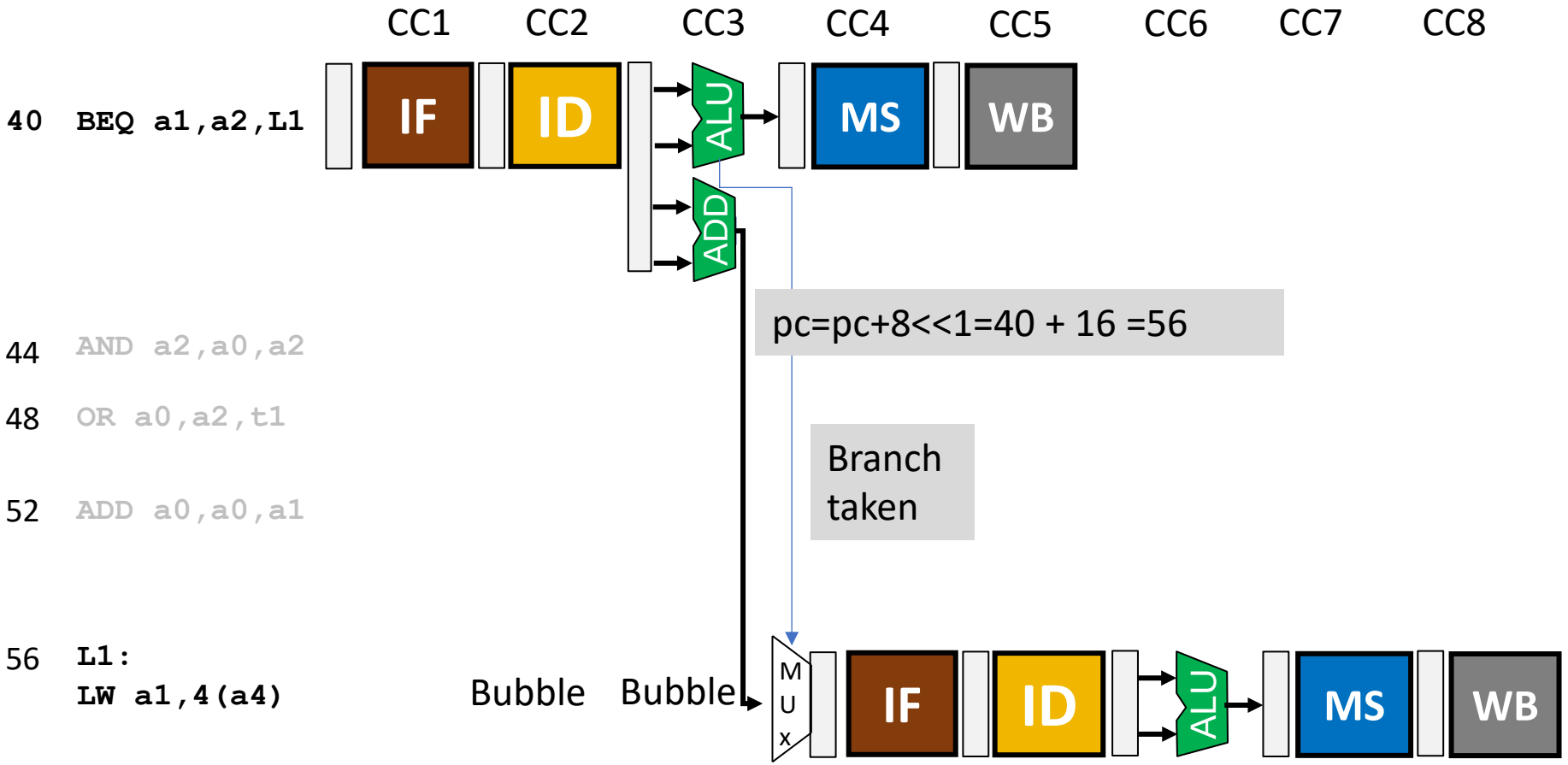
Handling Control Hazards: Conservative Approach (Branch not Taken)

- Control hazard (branch not taken): stall pipeline until decision known
- Branch penalty: 2 clock cycles



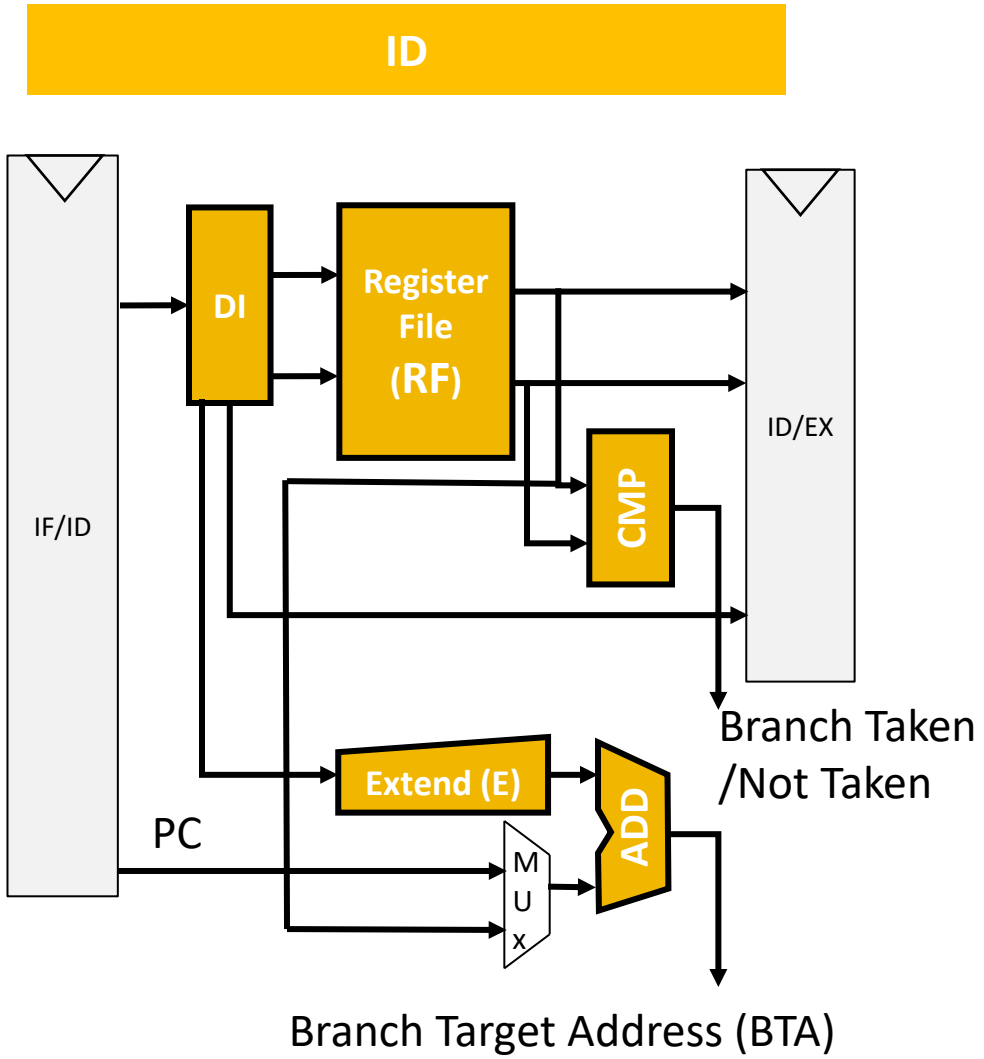
Handling Control Hazards: Conservative Approach (Branch Taken)

- Control hazard (branch taken): stall pipeline until decision and branch target known
- Branch penalty: 2 clock cycles



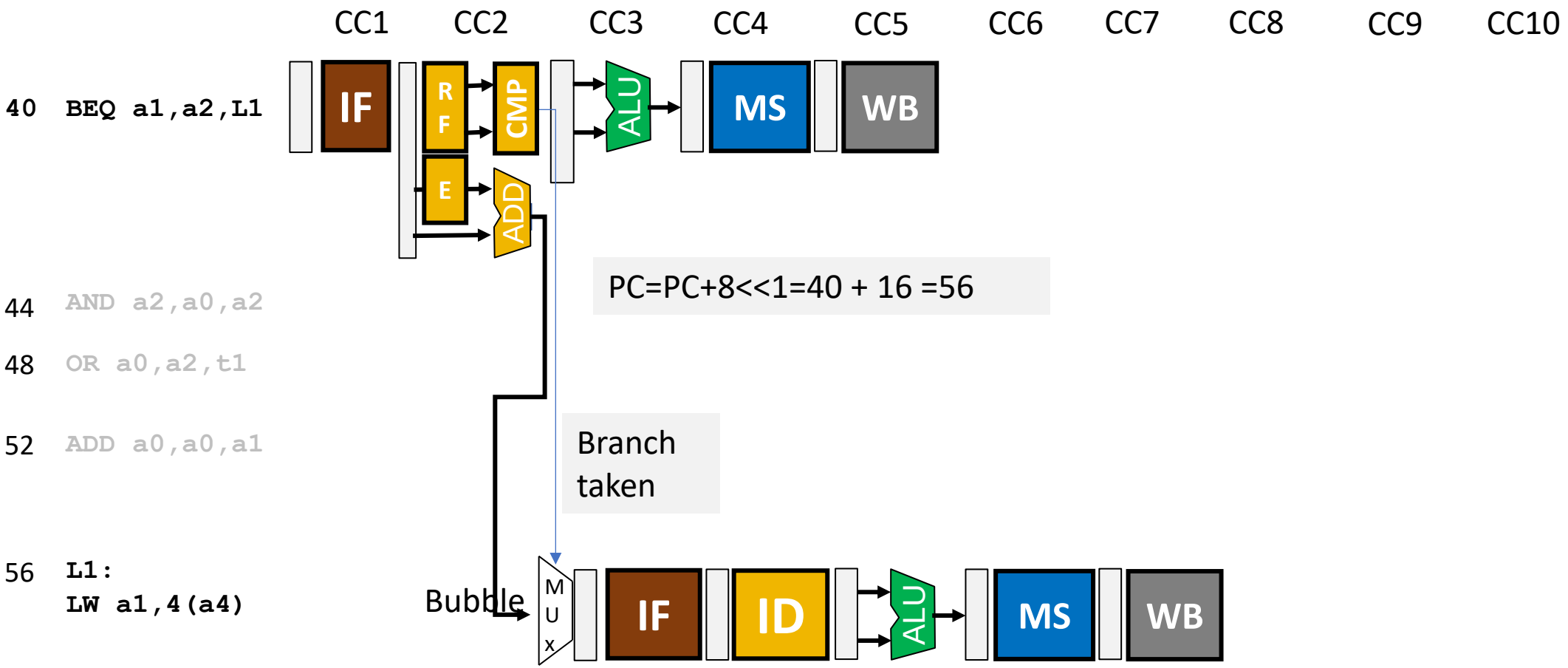
Reducing Branch Delay - Move Branch Decision to ID Stage

- **A lot of branches rely on simple tests (e.g., equality)**
- **Add hardware to determine outcome of branch in the ID stage**
  - → Reduce cost of the taken branch
  - Subcomputation: Compute Branch Target Address in ID
    - Move target address adder from EX to ID
    - PC and immediate are already in IF/ID pipeline register
  - Subcomputation: Comparison
    - Additional register comparator (done before in EX via the ALU)
    - **Additional Forwarding and Hazard Handling**



Reducing Branch Delay - Move Branch Decision to ID Stage – Branch Taken

- Target address adder in ID, Extra comparator to get branch decision in ID
- Branch penalty: Only one clock cycle



Static Branch Prediction

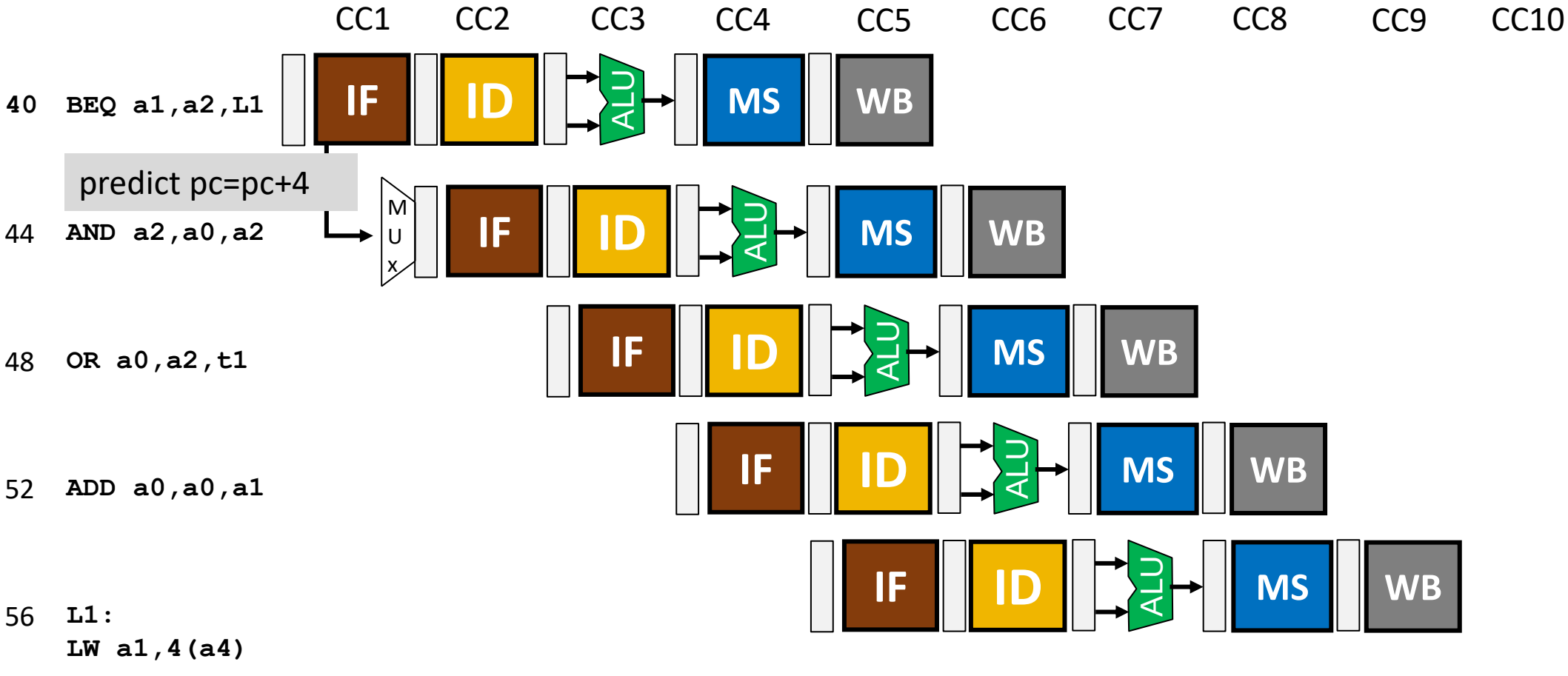
Motivation: Branch Prediction

- Longer pipelines can't readily determine branch outcome early
  - Branch penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong

- Simple Static Branch Prediction Schemes
  - **Always not Taken**: Always predict branches not taken – Also called fall through (PC=PC+4)
  - **Always taken**: Always predict branches taken

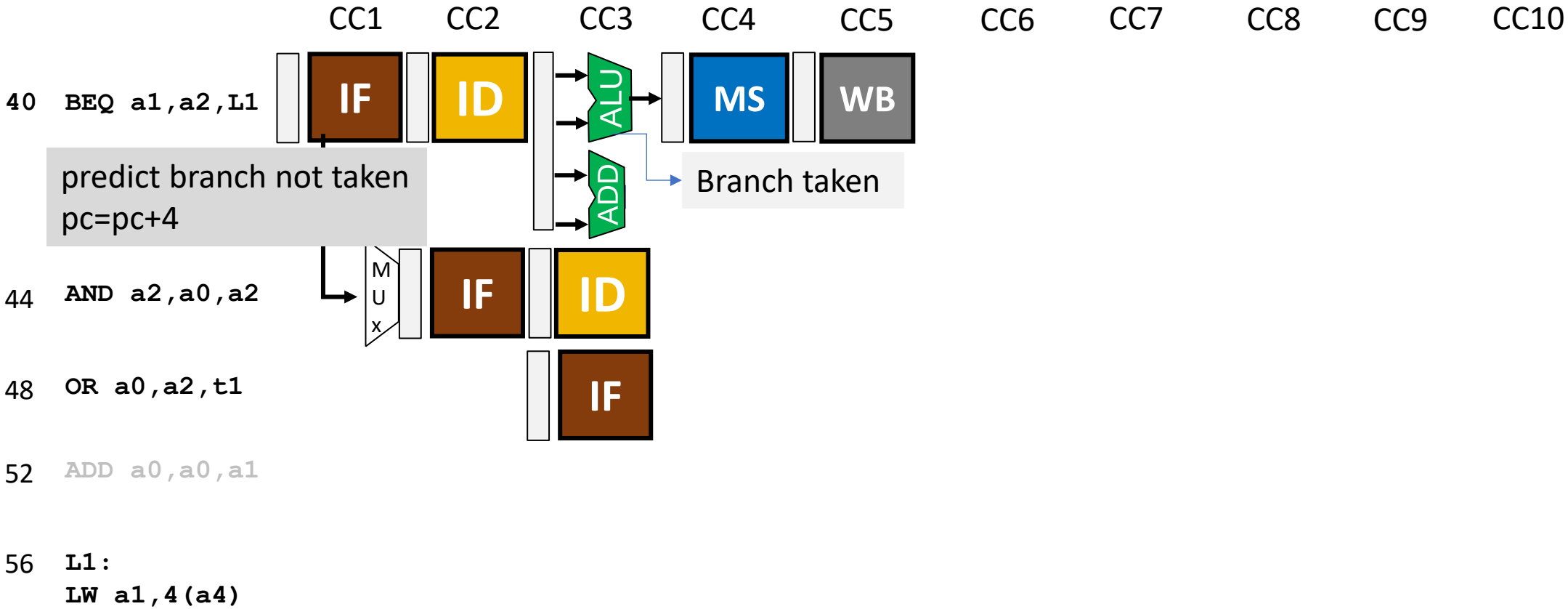
Always Not Taken – Correct Prediction

- Prediction correct (Branch not taken)
- Branch penalty: 0 clock cycles



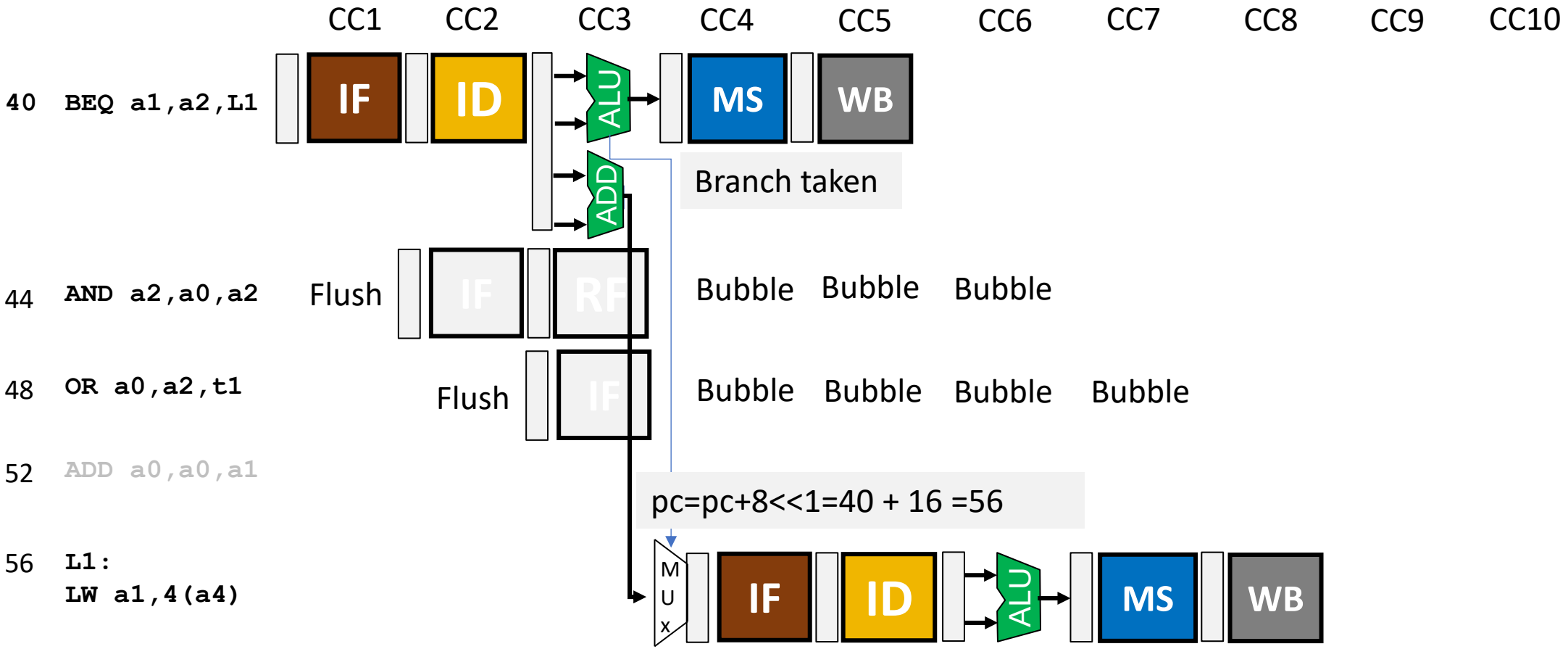
Always not Taken – Incorrect Prediction (1/2)

- Prediction incorrect (Branch not taken) – Flush instructions from pipeline
- Branch penalty: 2 clock cycles



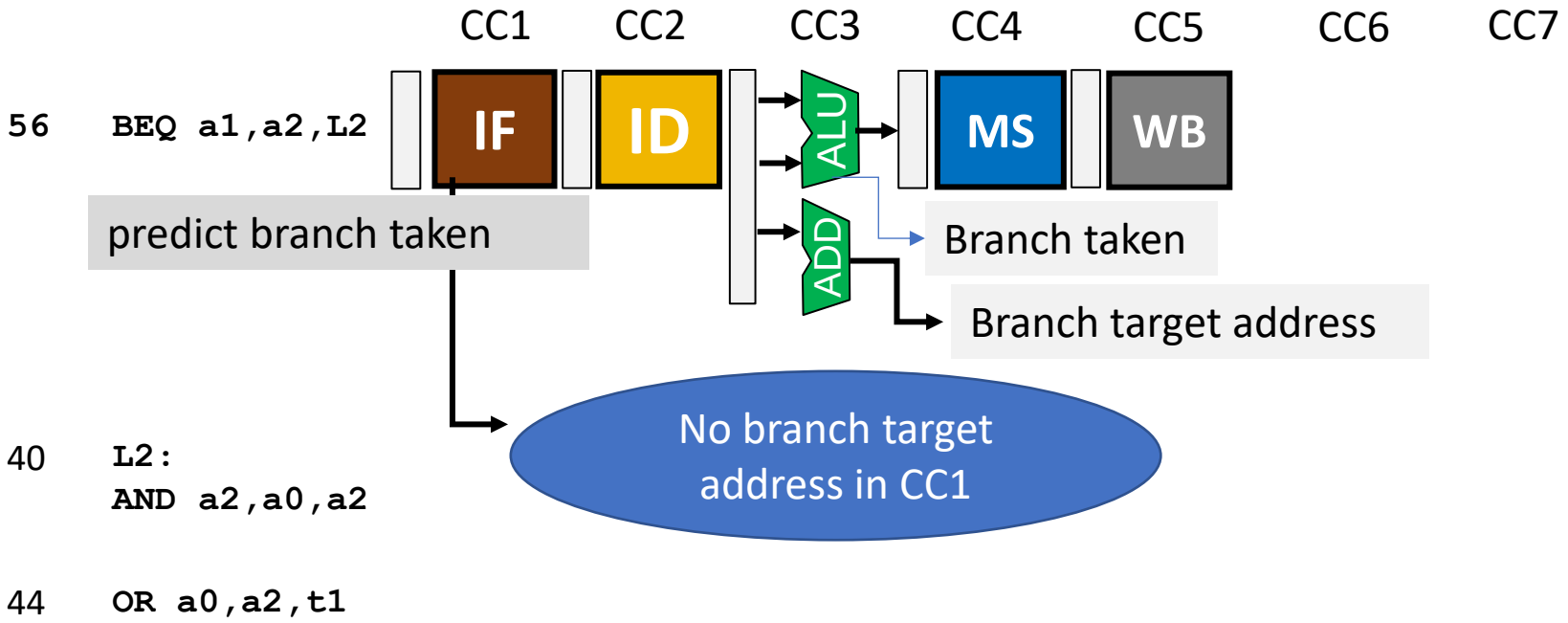
Always not Taken – Incorrect Prediction (2/2)

- Prediction incorrect (Branch not taken) – Flush instructions from pipeline
- Branch penalty: 2 clock cycles



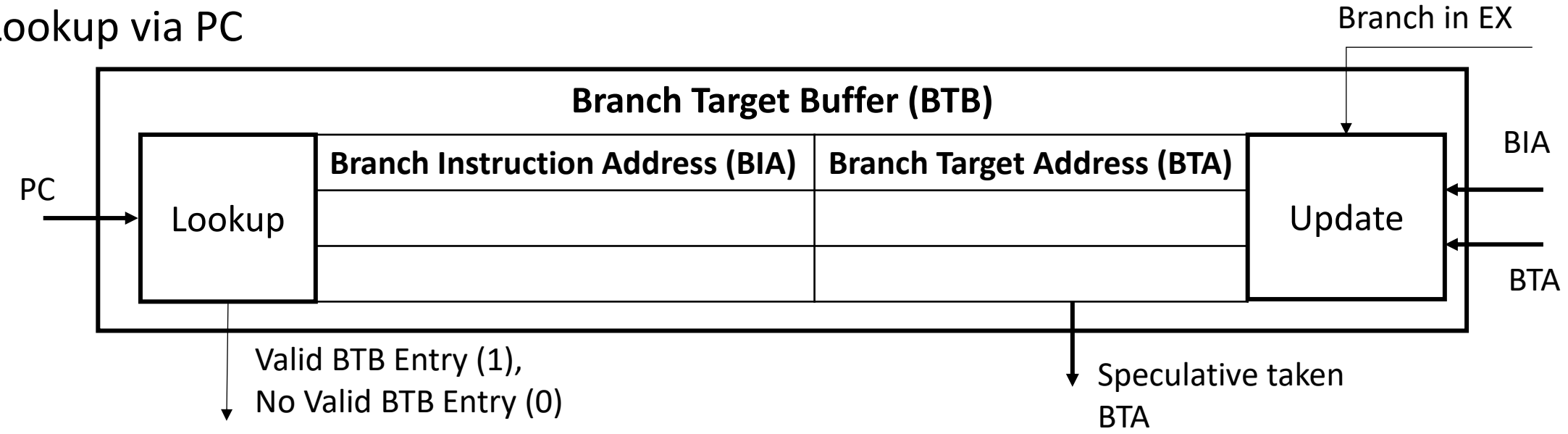
Always Taken – Correct Prediction

- Prediction (Branch taken) –> Branch target address is computed in EX stage



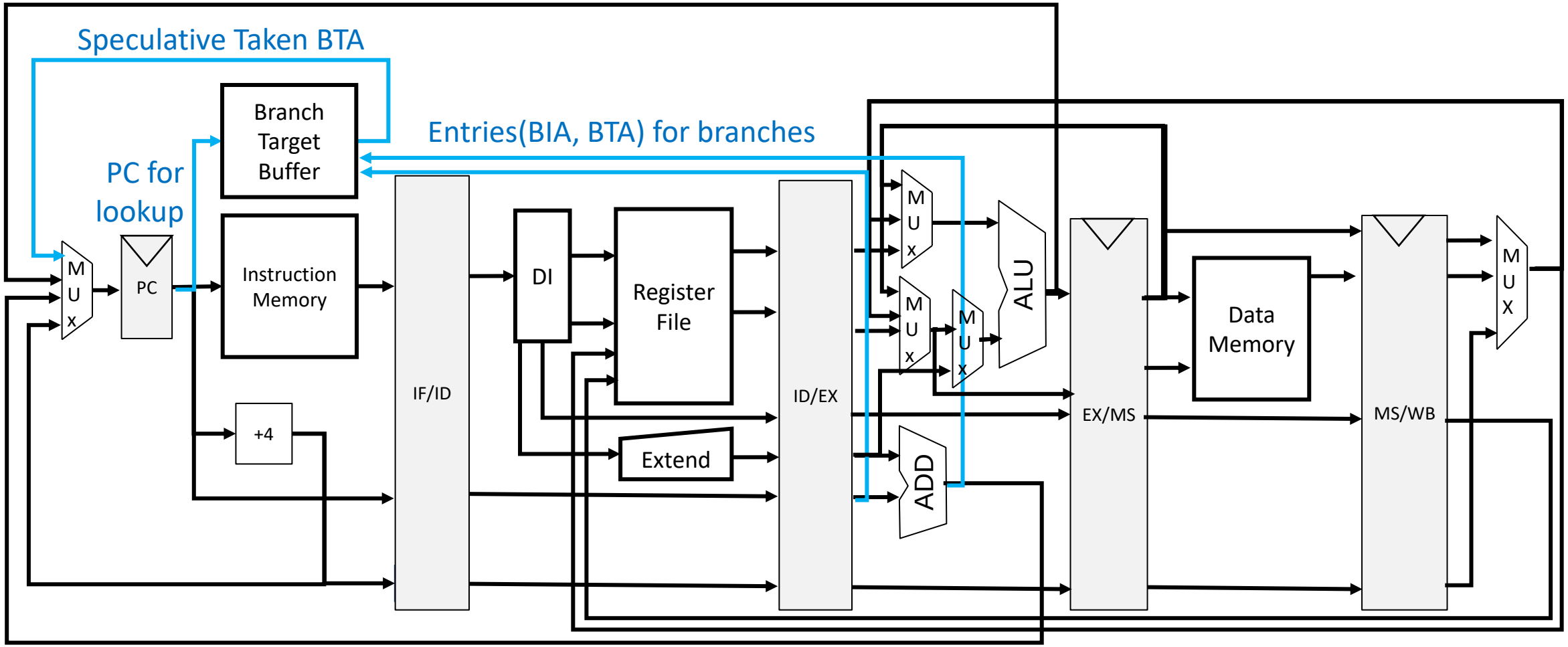
Branch Target Buffer (BTB)

- Stores the Branch Target Address (BTA) for a certain branch (e.g. identified by its own Branch Instruction Address (BTI))
- Content Addressable Memory (Costly for entries)
  - Update policy (similar to caches)
  - Entries entered in pairs (BIA, BTA)
  - entry not available for first branch execution
- Lookup via PC



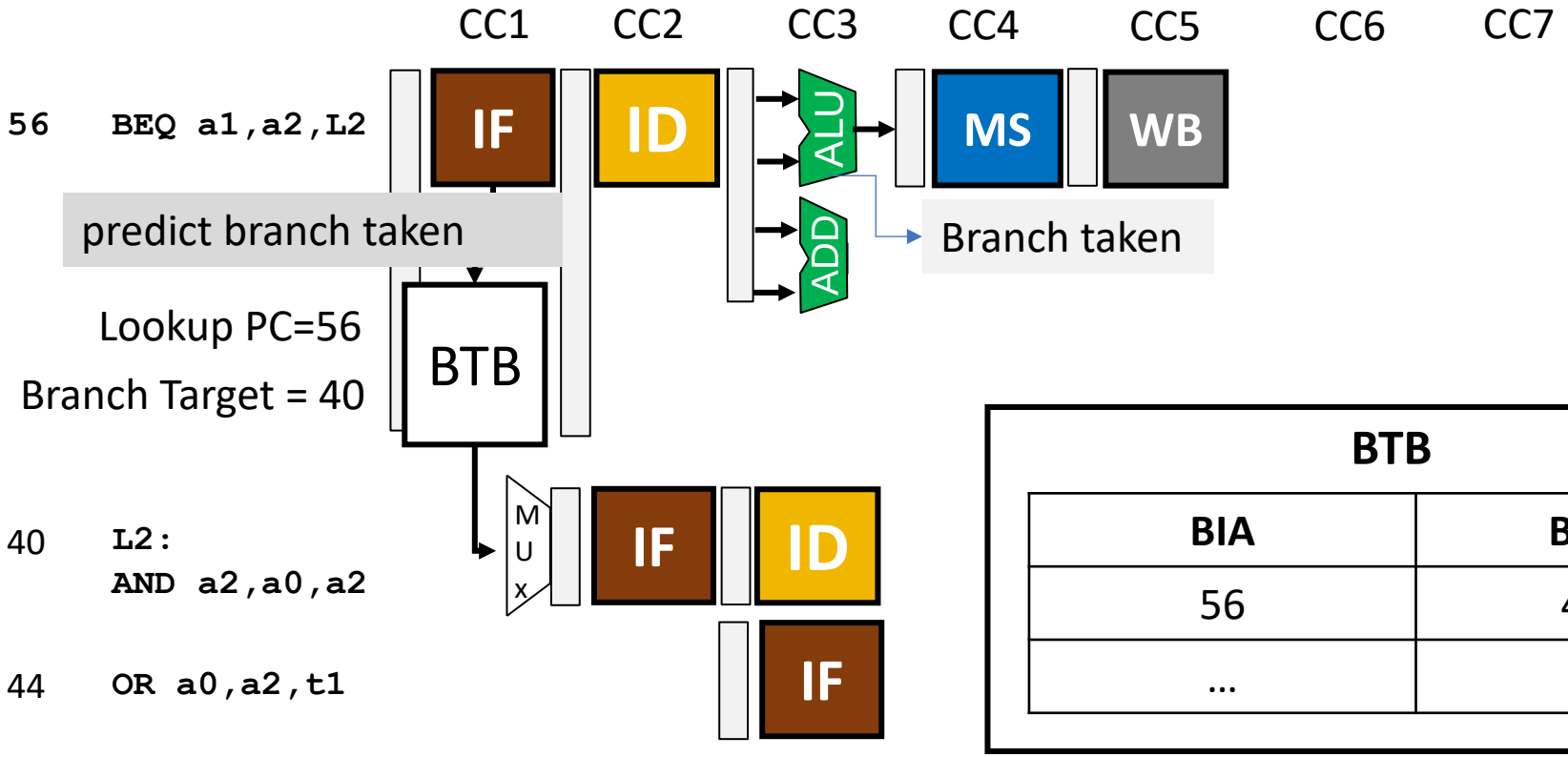
Five-stage Pipeline with Branch Target Buffer

- Only for branches and PC-relative Jumps J, JAL (not JALR, JR, RET)



Always Taken – Correct Prediction (BTB has entry)

- Prediction (Branch taken) - BTA via Branch Target Buffer (BTB)
- Branch penalty: 0 clock cycles



BTB

BIA	BTA
56	40
…	…

First execution of branch we cannot do a branch taken prediction.
Entry was written to BTB on earlier execution of branch with (56,40)

BTFNT: Enhancing Static Branch Prediction

Typical Statistics 60% to 70% of branches are taken

Example:

- 60% are backward branches (negative offset)
  - Loops: Usual more than one iteration (branch will be taken more than once) – taken ~90%
  - Typical behavior: T T T T…T NT
  - About 90% of backward branches are taken

- 40% are forward branches (positive offset)
  - If-(Else) Constructs: Branches go forward (jump over code)
  - About ~20% of forward branches are taken

- Always not taken: $(0{,}6 \cdot 0{,}9) + (0{,}4 \cdot 0{,}2) = 62\%$ mispredictions
- Always taken: $(0{,}6 \cdot 0{,}1) + (0{,}4 \cdot 0{,}8) = 38\%$ mispredictions
- **Enhanced Static Branch Prediction: Backward Taken, Forward Not Taken (BTFNT)**
  - Predict **forward** branches **not taken: ~10% mispredictions**
  - Predict **backward** branches **taken: ~20% mispredictions**
  - **Overall:** $(0{,}6 \cdot 0{,}1) + (0{,}4 \cdot 0{,}2) = 14\%$ mispredictions

Effect of Misprediction Rate and Branch Penalty on CPI

Program with:

- Relative number of branch instructions (branch rate ***b***)
- The branch cycle penalty ***p*** for mispredictions
- The branch misprediction rate ***m***

- **CPI**: Cycles per Instructions (data hazards rare so base CPI=1)

$$\textbf{CPI} = 1 + b \cdot p \cdot m$$

Five stage pipeline: ***b***=15%, ***p***=2
Always not taken: ***m***=62% -> **CPI** = 1,186
Always taken: ***m***=38% -> **CPI** = 1,114

BTFNT: ***m***= 14% -> **CPI** = 1,042

Longer pipeline: ***b***=15%, ***p***=5
Always not taken: ***m***=62% -> **CPI** = 1,465
Always taken: ***m***=38% -> **CPI** = 1,285

BTFNT: ***m***= 14% -> **CPI** = 1,105

In **longer** pipelines, branch penalty is more significant

Dynamic Branch Prediction

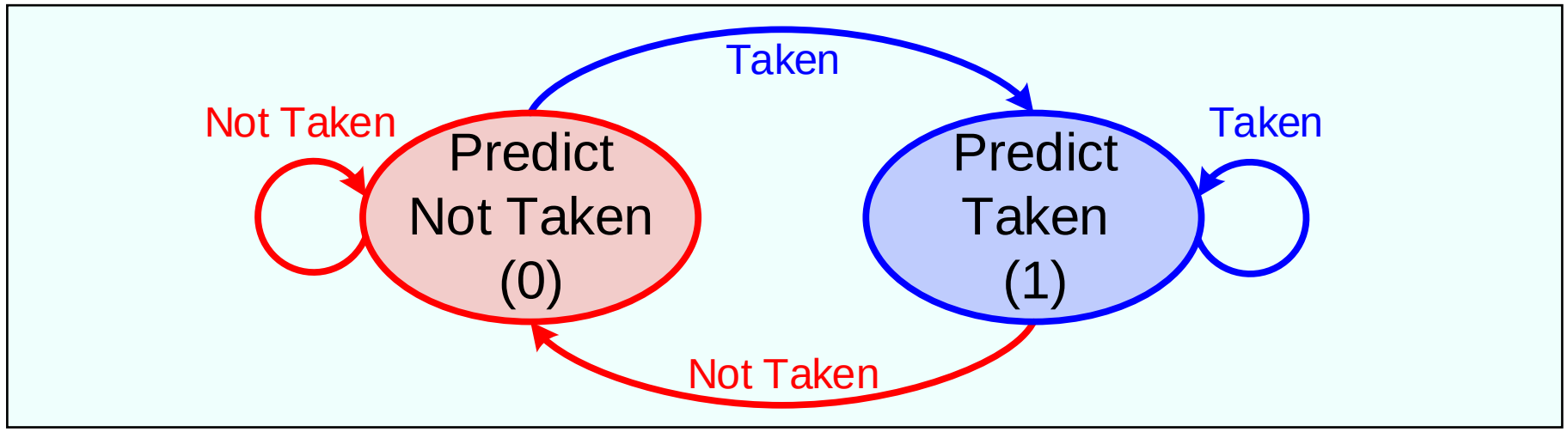
Dynamic Branch Prediction

- In **longer** pipelines, branch penalty is more significant

- *Branch prediction buffer* (aka branch history table (BHT)) for dynamic prediction
  - Stores last outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through (not taken) or target (taken)
    - In case of misprediction, flush pipeline and flip prediction

Dynamic Branch Prediction: 1-Bit Predictor

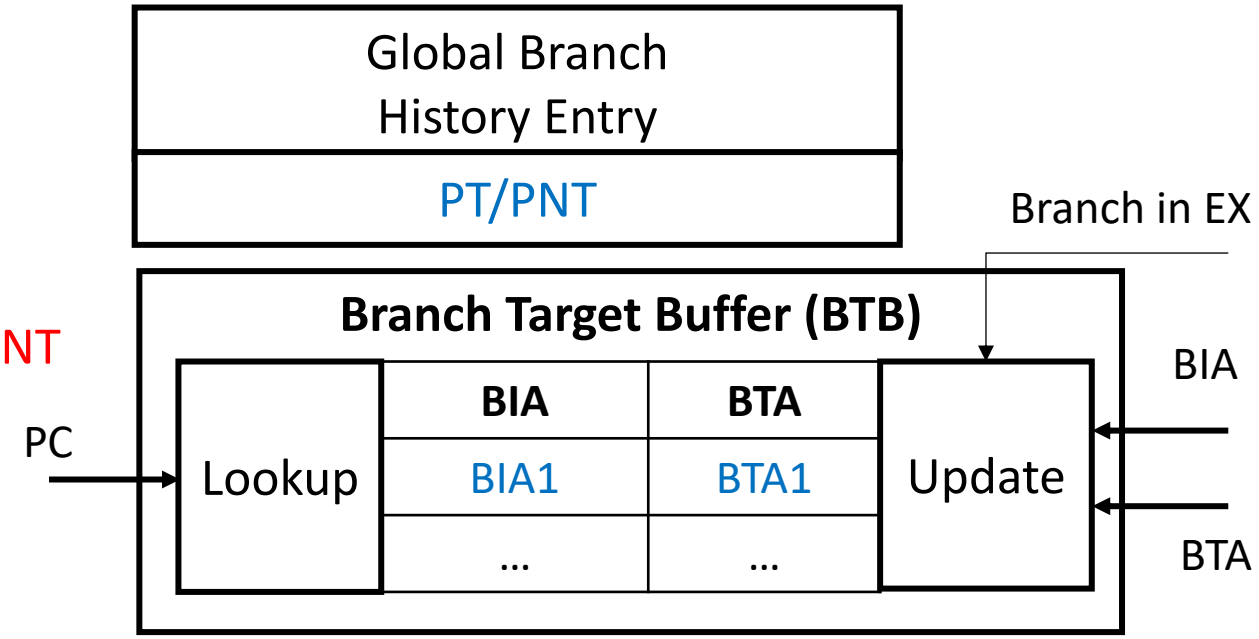
- **Single-Bit / 1-Bit / Last-Time Predictor**
  - Indicates *which direction the branch went last time it executed*
  - PNT: Predict NT (Bit=0): Fetch the instruction from (PC+4)
  - PT: Predict T (Bit=1): Get target address from the BTB



Global Predictor

- One single Branch History Entry for all branches to save last decision

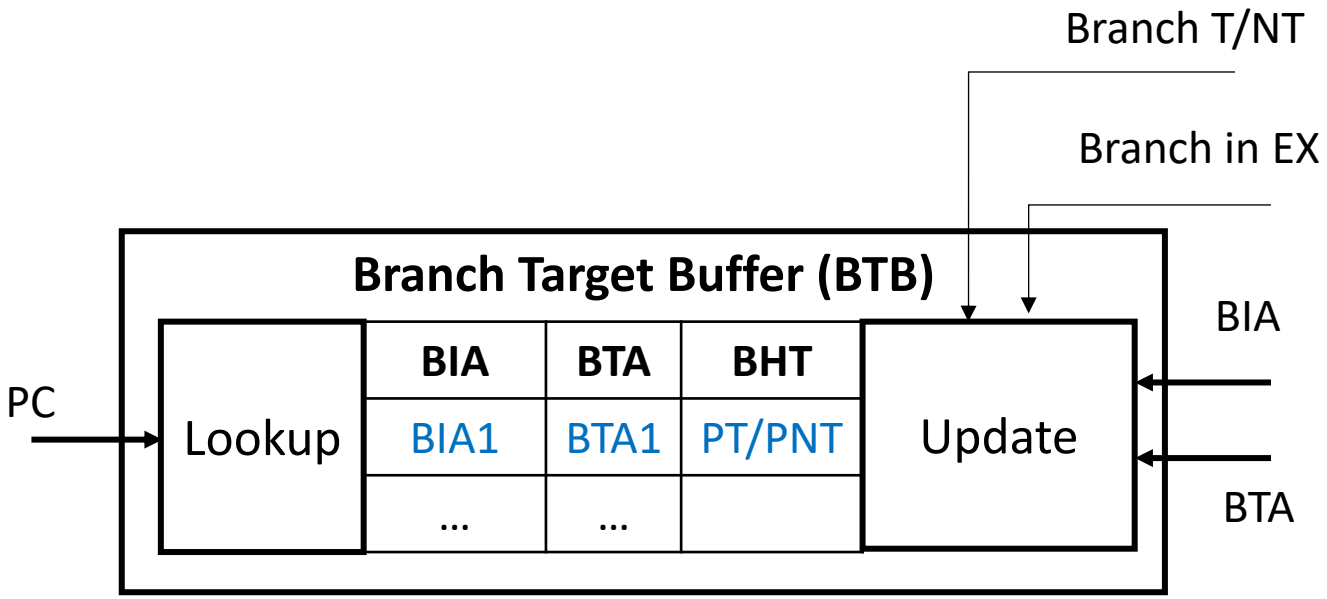
- Branch reaches IF stage
  - Indexed Lookup with PC in BTB
  - No valid BTB entry
    –> predict NT (PNT)
    -> Supply **PC=PC+4**
  - Valid BTB entry
    -> Global Predictor result based on BHT: PT/PNT
    -> Supply **PC=BTA/PC+4**
- Branch reaches EX stage
  - Indexed Lookup with PC in BTB
    -> No BTB entry -> Update BTB (create entry)
    -> Eventually only in case that branch is taken
  - Update Global Branch History Entry



Local Predictor

- Branch History Table (BHT): One entry for each BTB entry

- Branch in IF stage
  - Indexed Lookup with PC in BTB
  - No valid BTB entry
    –> predict NT
    -> Supply **PC=PC+4**
  - Valid BTB entry
    -> Local BHT Predictor result T/NT
    -> Supply **PC=BTA/PC+4**
- Branch in EX stage
  - Indexed Lookup with BIA in BTB
    - No BTB entry -> Update BTB (create entry), initialize BHB with T/NT
    - BTB entry: Update Local BHT with T/NT



Example Nested Loop Program - Static Branch Prediction

- Example Nested Loop Program:

```
for (x = 1024; x > 0; x--)
  for (y = 4; y > 0; y--)
    do_something(x,y);
```

```
01:   li s0, 1024
02: xloop:
03:    li s1, 4
04: yloop:
05:   mv a0, s0
06:   mv a1, s1
07:   jal ra, do_something
08:   addi s1, s1, -1
09:   bnez s1, yloop
10:   addi s0, s0, -1
11:   bnez s0, xloop
```

Inner Loop (L09 Branch Pattern): (T-T-T-NT)
Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

Static Branch Prediction:

- **Always not taken:** ~80% Mispredictions
- **Always taken:** ~20% Mispredictions
- **BTFNT (same as always taken):** ~20% Mispredictions

Example Nested Loop Program - Dynamic Branch Prediction (1bit Global)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  N=No, Y=Yes
  Misprediction rate:

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y												
BTB entry L11	Y												
Global BHT	PNT												
Prediction	NT												
Direction	-												
Correct?	-												

Example Nested Loop Program - Dynamic Branch Prediction (1bit Global)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  N=No, Y=Yes
  Misprediction rate:

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y	Y	Y	Y	Y	Y							
BTB entry L11	Y	Y	Y	Y	Y	Y							
Global BHT	PNT	PNT	PT	PT	PT	PNT							
Prediction	NT	NT	T	T	T	NT							
Direction	-	T	T	T	NT	T							
Correct?	-	*N*	*Y*	*Y*	*N*	*N*							

Example Nested Loop Program - Dynamic Branch Prediction (1bit Global)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  N=No, Y=Yes
  Misprediction rate: ~40% (2 out of five)

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Repeats

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	…	…
BTB entry L11	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	…	…
Global BHT	PNT	PNT	PT	PT	PT	PNT	PT	PT	PT	PT	PNT	…	…
Prediction	NT	NT	T	T	T	NT	T	T	T	T	NT	…	…
Direction	-	T	T	T	NT	T	T	T	T	NT	T	…	…
Correct?	-	*N*	*Y*	*Y*	*N*	*N*	*Y*	*Y*	*Y*	*N*	*N*	…	…

Example Nested Loop Program - Dynamic Branch Prediction (1bit Local)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Misprediction rate:

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y												
BTB entry L11	Y												
BHT L9	PNT												
BHT L11	PNT												
Prediction	PNT												
Direction	-												
Correct?	-												

Example Nested Loop Program - Dynamic Branch Prediction (1bit Local)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  Misprediction rate: ~40% (2 out of five)

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y	Y	Y	Y	Y	Y							
BTB entry L11	Y	Y	Y	Y	Y	Y							
BHT L9	PNT	PNT	PT	PT	PT	PNT							
BHT L11	PNT	PNT	PNT	PNT	PNT	PNT							
Prediction	PNT	NT	T	T	T	NT							
Direction	-	T	T	T	NT	T							
Correct?	-	*N*	*Y*	*Y*	*N*	*N*							

Example Nested Loop Program - Dynamic Branch Prediction (1bit Local)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Misprediction rate: ~40% (2 out of five)

Repeats

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	…	…
BTB entry L11	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	…	…
BHT L9	PNT	PNT	PT	PT	PT	PNT	PNT	PT	PT	PT	PNT	…	…
BHT L11	PNT	PNT	PNT	PNT	PNT	PNT	PT	PT	PT	PT	PT	…	…
Prediction	PNT	NT	T	T	T	NT	NT	T	T	T	T	…	…
Direction	-	T	T	T	NT	T	T	T	T	NT	T	…	…
Correct?	-	*N*	*Y*	*Y*	*N*	*N*	*N*	*Y*	*Y*	*N*	*Y*	…	…

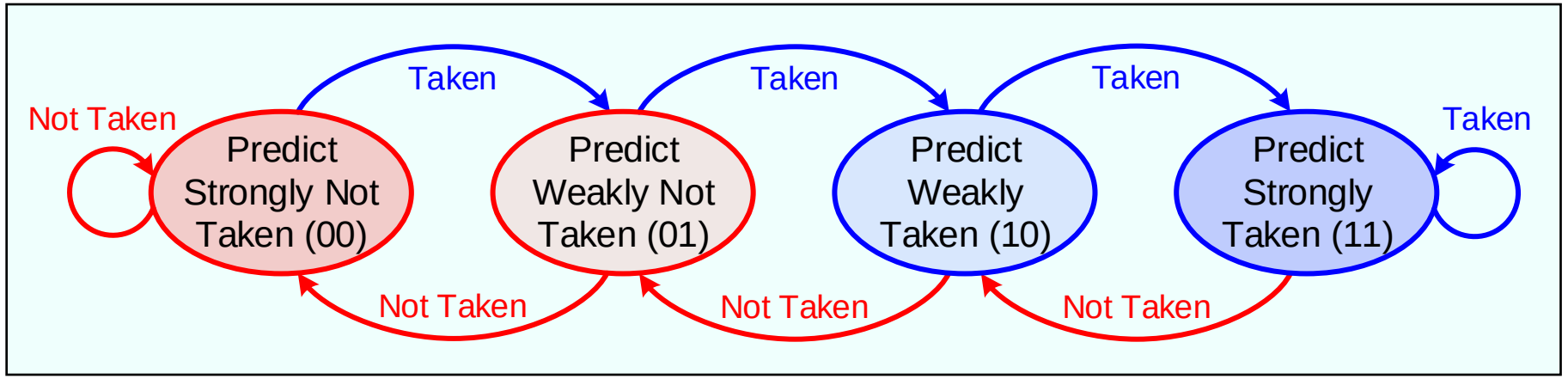
Improving the 1-Bit Predictor

- **Problem:** A 1-bit 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

2-Bit Predictor

- Prediction does not change on a single misprediction
- 2-Bit entry in BHT => Four States [2 for NT, 2 for T]
  - PSNT: Strongly Not Taken (00), PWNT: Weakly Not Taken (01)
  - PWT: Weakly Taken (10), PST: Strongly Taken (11)
- 2-Bit Counter
  - Increment by 1 if branch taken, otherwise decrement by 1
  - Saturate the counter value at 0 and 3
  - **A prediction must be wrong twice (consecutively) before the prediction bit is changed**



Example Nested Loop Program - Dynamic Branch Prediction (2bit Global)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  N=No, Y=Yes
  Misprediction rate:

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y												
BTB entry L11	Y												
Global BHT	PWNT												
Prediction	NT												
Direction	-												
Correct?	-												

Example Nested Loop Program - Dynamic Branch Prediction (2bit Global)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  N=No, Y=Yes
  Misprediction rate:

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y	Y	Y	Y	Y								
BTB entry L11	Y	Y	Y	Y	Y								
Global BHT	PWNT	PWNT	PWT	PST	PST								
Prediction	NT	NT	T	T	T								
Direction	-	T	T	T	NT								
Correct?	-	*N*	*Y*	*Y*	*N*								

Example Nested Loop Program - Dynamic Branch Prediction (2bit Global)

- Example Nested Loop Program:

  Inner Loop (L09 Branch Pattern): (T-T-T-NT)
  Nested Loop Pattern: (T-T-T-NT) T (T-T-T-NT) T (T-T-T-NT) T….

  N=No, Y=Yes
  Misprediction rate: ~20% (1 out of five)

```
01:  li s0, 1024
02: xloop:
03:   li s1, 4
04: yloop:
05:  mv a0, s0
06:  mv a1, s1
07:  jal ra, do_something
08:  addi s1, s1, -1
09:  bnez s1, yloop
10:  addi s0, s0, -1
11:  bnez s0, xloop
```

Repeats

Branch	**Start**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**	**L09**	**L09**	**L11**	**L09**	**L09**
BTB entry L09	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	…	…
BTB entry L11	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	…	…
Global BHT	PWNT	PWNT	PWT	PST	PST	PWT	PST	PST	PST	PST	PWT	…	…
Prediction	NT	NT	T	T	T	T	T	T	T	T	T	…	…
Direction	-	T	T	T	NT	T	T	T	T	NT	T	…	…
Correct?	-	*N*	*Y*	*Y*	*N*	*Y*	*Y*	*Y*	*Y*	*N*	*Y*	…	…

2-bit Predictor: Limits

- Still penalty on regular patterns:
  - Recap: Inner loop iterations: T-T-T-NT
  - Branches often show such regular patterns
- Can we incorporate this regularity? -> Use a history

- Two-level-history adaptive branch predictors (many variants *)
  - Learn the history and loop pattern T-T-T-NT
  - They usually can have higher accuracy
  - This is still 90ties technology *

*Tse-Yu Yeh and Y. N. Patt, "A Comparison Of Dynamic Branch Predictors That Use Two Levels Of Branch History," *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, CA, USA, 1993

- Modern branch predictors for complex processors
  - Based on neural networks
  - Learn patterns, history and interrelation between branches
  - Can achieve very small misprediction rates

Optional, not relevant for exam

A Look at a Real Processor – CVA6

“CVA6 is a RISC-V compatible application processor core that can be configured as a 32- or 64-bit core: **CV32A6** and **CV64A6”.**

--- CVA& User Manual

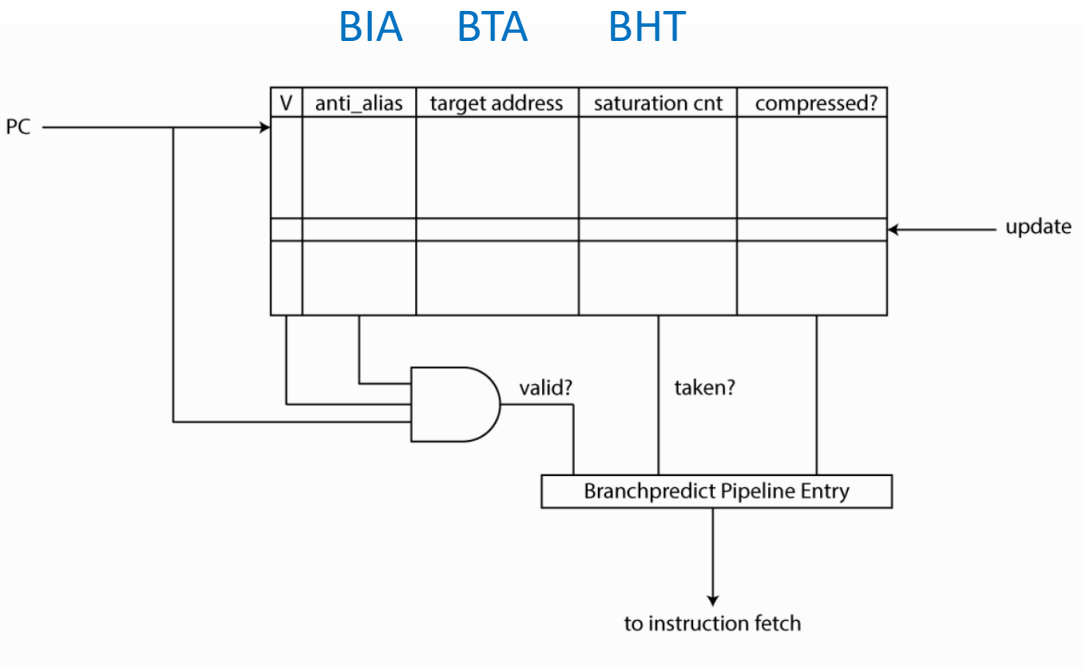
https://docs.openhwgroup.org/projects/cva6-user-manual/01_cva6_user/Introduction.html

Developed initially as part of PULP project (ETH Zürich), now maintained by the OpenHW Group

CVA6 Branch Predictor

"**Branch Predict**: If the BHT and BTB predict a branch on a certain PC, PC Gen sets the next PC to the predicted address and also informs the IF stage that it performed a prediction on the PC. (…)"

„All branch prediction data structures reside in a single register-file like data structure. It is indexed with the appropriate number of bits from the PC and contains information about the predicted target address as well as the outcome of a **configurable-width saturation counter (two by default).** The prediction result is used in the subsequent stage to jump (or not)."



-- CVA6 Design Document (deprecated) – Branch Prediction (05.04.2024)
https://docs.openhwgroup.org/projects/cva6-user-manual/03_cva6_design/pcgen_stage.html

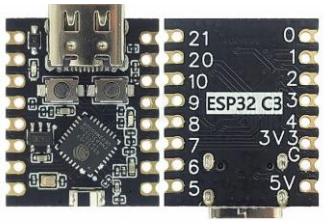
Optional, not relevant for exam

A Look at a Real Processor – ESP32-C3

ESP32-C3 Technical Reference Manual

https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf#riscvcpu

Low Power Mikro-Controller – ESP32-C3



Picture: Alibaba-
Costs less than 1€

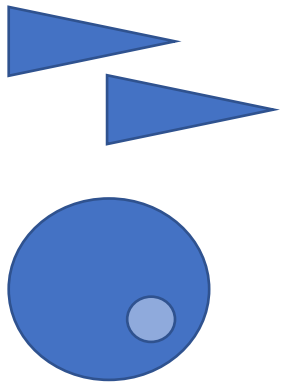
Scalar in-order processors with five or less pipeline stages are used in low-cost **micro-controller-type** devices.

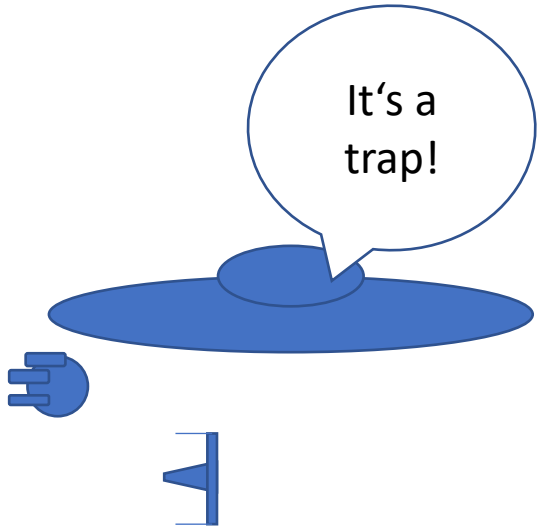
„ESP-RISC-V CPU is a 32-bit core based upon RISC-V ISA comprising base integer (I), multiplication/division (M) and compressed (C) standard extensions. The core has **4-stage, in-order, scalar pipeline** optimized for area, power and performance. (…)“

-- ESP32-C3 Technical Reference Manual
https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf#riscvcpu

Optional, not relevant for exam

Trap Handling





Terminology

- Terminology is used often different for different architectures (x86,ARM, RISCV,…).

- For RISC-V:
- “We use the term **exception** to refer to an unusual condition occurring at run time associated with an instruction in the current RISC-V hart.”
- “We use the term **interrupt** to refer to an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control.”
- “We use the term **trap** to refer to the transfer of control to a trap handler caused by either an exception or an interrupt.”

—- Volume 1, Unprivileged Specification version 20191213:
https://riscv.org/technical/specifications/

Trap Handling

- For a function call the compiler assures that the function call standard of the ABI is kept
- An exception and interrupt can happen during execution of a function ceither due to an instruction (e.g. memory access error) or due to an external event (device raises an interrupt)
- For a trap, we are in the middle of execution of a function and must save the context of the current execution before calling a trap handler to handle the exception or interrupt
- RISC-V has certain so-called **Control Status Registers (CSRs)** to identify the cause of a trap

Trap Handling Basics

- Trap is detected.
- Change mode
- Jump to trap handler
- Trap handler saves context
- Trap handler identifies cause (exception/interrupt)
- Corresponding exception/interrupt handler is called
  - Some handlers do not return if they can not recover from an exception
- Trap handler restores context
- Change mode
- Jump back to program execution

Causes for Traps

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

- —- Volume 2, Privileged Specification version 20211203: https://riscv.org/technical/specifications/

  Word trap is mentioned 301 times

- Different architectures treat exceptions differently e.g. division by zero is not raising an exception in RISC-V

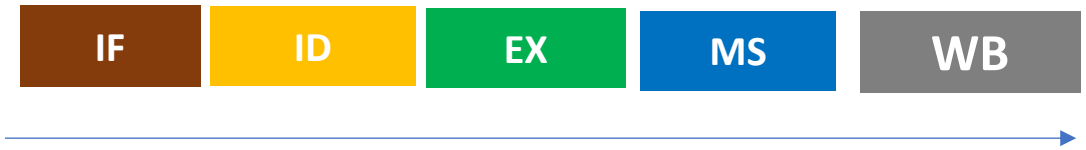
Precise vs. Imprecise traps

- Precise traps:
  - Associated with a certain instruction (e.g. illegal instruction exception)
  - Easier to debug
- Imprecise trap:
  - Not associated with an instruction
  - Hard to debug
  - OR: Pipelined execution makes it hard to associate the exception with an instruction (This is an issue with certain pipelines, which we see in next lecture)

Summary

Where we are

- Five-Stage Scalar In-order Processor Pipeline
  - Forwarding to mitigate data hazards
  - Branch prediction to mitigate control hazards



- In-order pipeline
- Five Stages
- Scalar pipeline: CPI >= 1

- Upcoming Lecture: Multi-cycle Functional Units (DIV/MUL) and Out-of-Order (OoO)

Thank you for your attention!

BACKUP

Registers of RISC-V

- RISC-V has 32 integer registers
- Processors can have different register width, we look at RV32 with 32-bit width
- Each register has two IDs (x0-x31) and **an ABI name** that indicates its role
- ABI stands for Application Binary Interface (ABI)

Register	ABI Name	Description	Saver
x0	Zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5-7	**t0-2**	**Temporaries**	**Caller**
x8	s0,fp	Saved register/frame pointer	Callee
x9	s1	Saved Register	Callee
x10-11	**a0-1**	**Function arguments, Return values**	**Caller**
x12-17	**a2-7**	**Function arguments**	**Caller**
x18-27	s2-11	Saved registers	Callee
x28-31	**t3-6**	**Temporaries**	**Caller**

Application Binary Interface (ABI) – Function Call Convention

- ABI also specifies rules for register usage in passing arguments and results for function calls
  - **Callee-saved registe**rs: If function foo1 (caller) calls foo2 (callee), then foo2 is not allowed to modify this value (it needs to save it and restore it before returning to foo1)
  - **Caller-saved registers**: If function foo1 (caller) calls foo2 (callee), then foo1 needs to save this register before calling foo2 if it wants to keep the value in it because foo1 is allowed to modify it

- According to ABI parameters are passed to a function in registers a0-a7
- The function should return its return value in register a0 (if <=32-bit value)

RISC-V Instructions

- The RISC-V ISA is modular with base instruction sets and a large variation of extensions
  - We look at **RV32IM**
- 32-bit Integer Instruction Set RV32I
  - Integer Register-Register Instructions (R-type)
    - Runs an arithmetic or logical operation on registers
    - Both operands are values in registers
  - Integer Register-Immediate Instructions (I-type)
    - Second operand is an immediate (constant) value
  - Control Transfer Instructions
    - Unconditional jumps
    - Conditional Branches
  - Load Store Instructions
    - Move data between memory and registers
    - Load-store Architecture: Operations on registers only
- 32-bit Integer Multiplication RV32M Extension -> Next Session
  - Integer Multiplication Instructions
  - Integer Division Instructions