CPE 221: Computer Organization

16 Pipelined ARM Processor

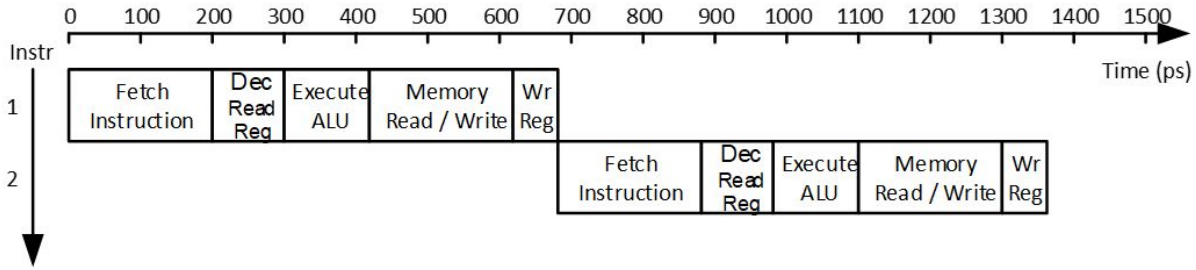
rahul.bhadani@uah.edu

Pipelined ARM Processor

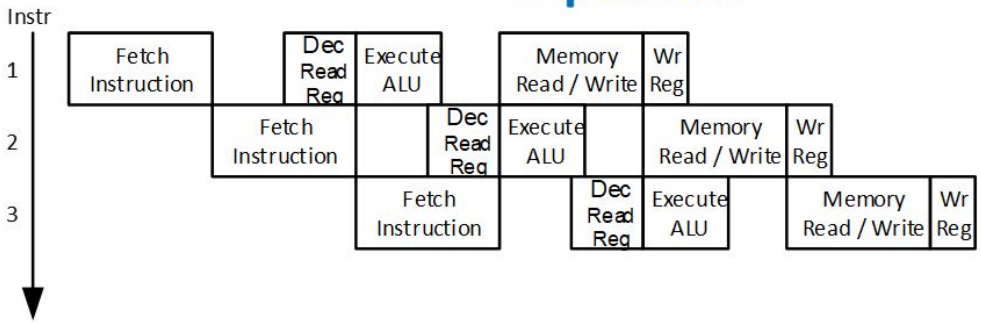
- Temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages

Single-Cycle vs. Pipelined

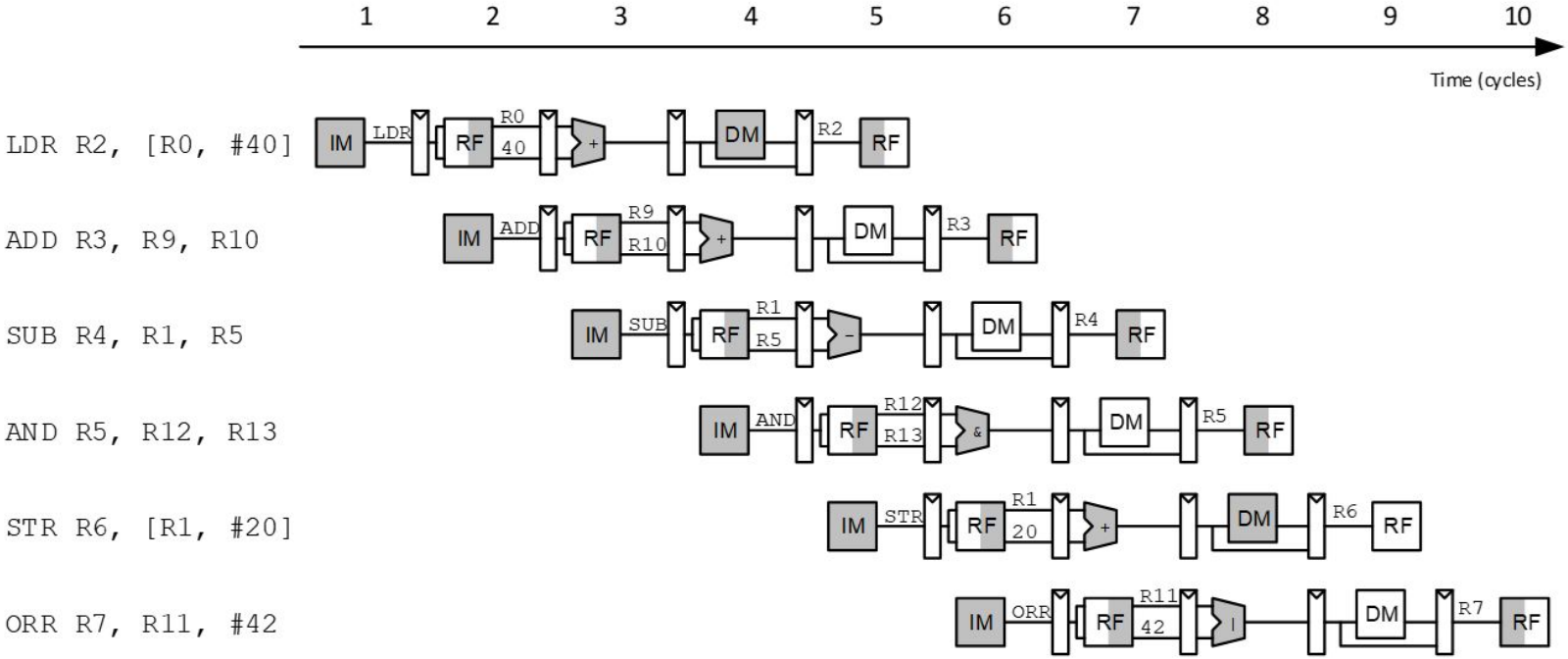
Single-Cycle



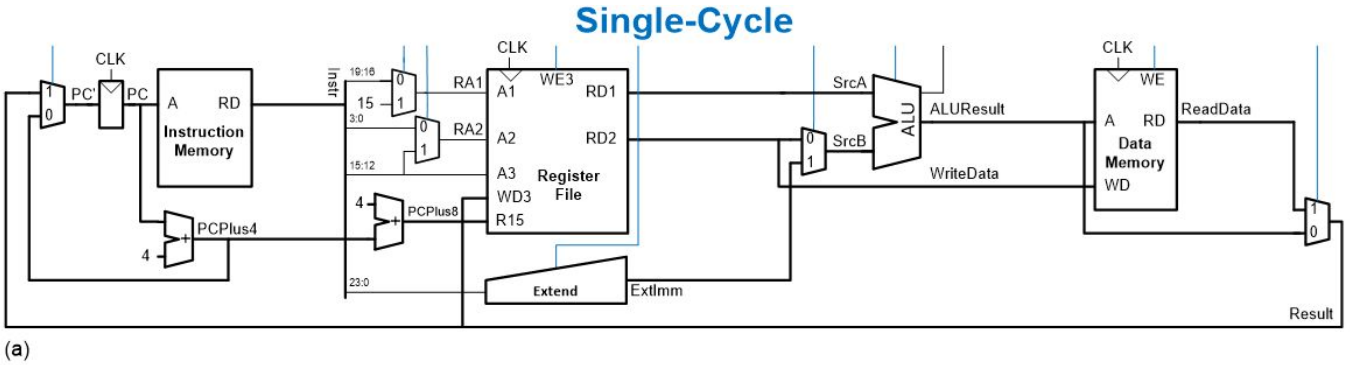
Pipelined

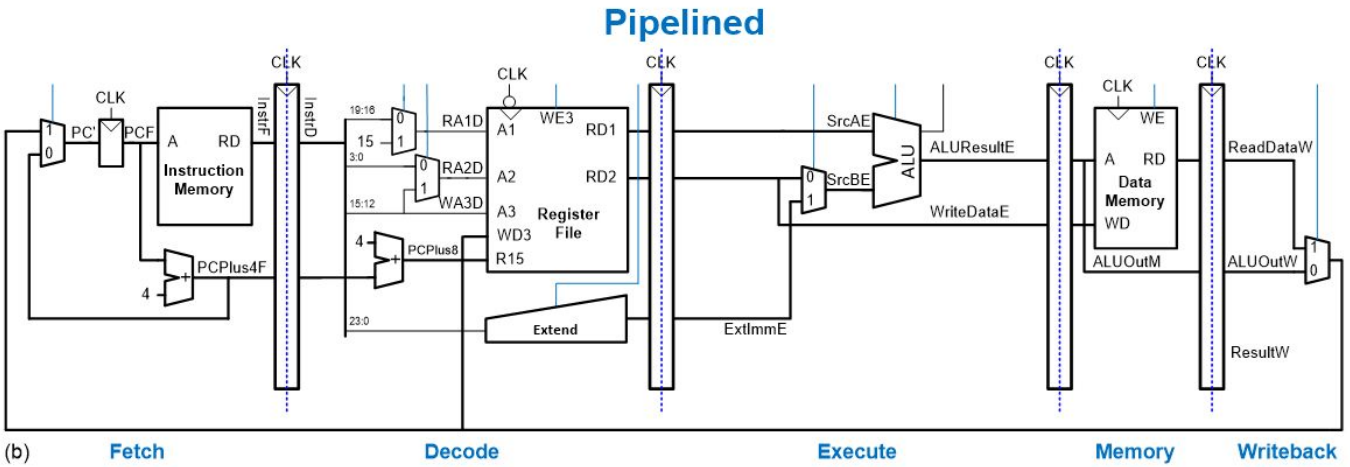


Pipelined Processor Abstraction

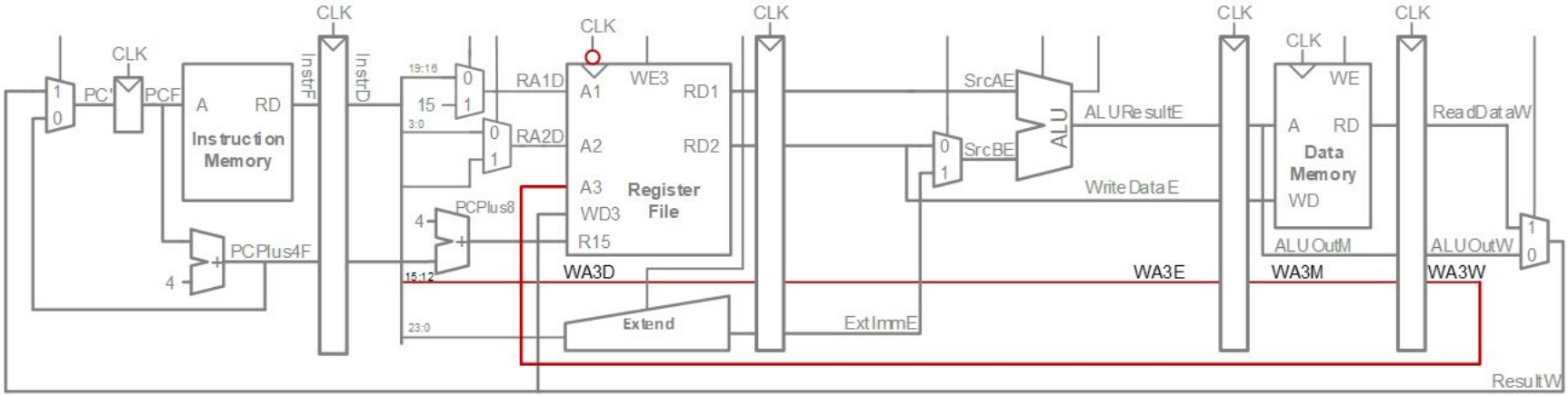


Single-Cycle & Pipelined Datapath





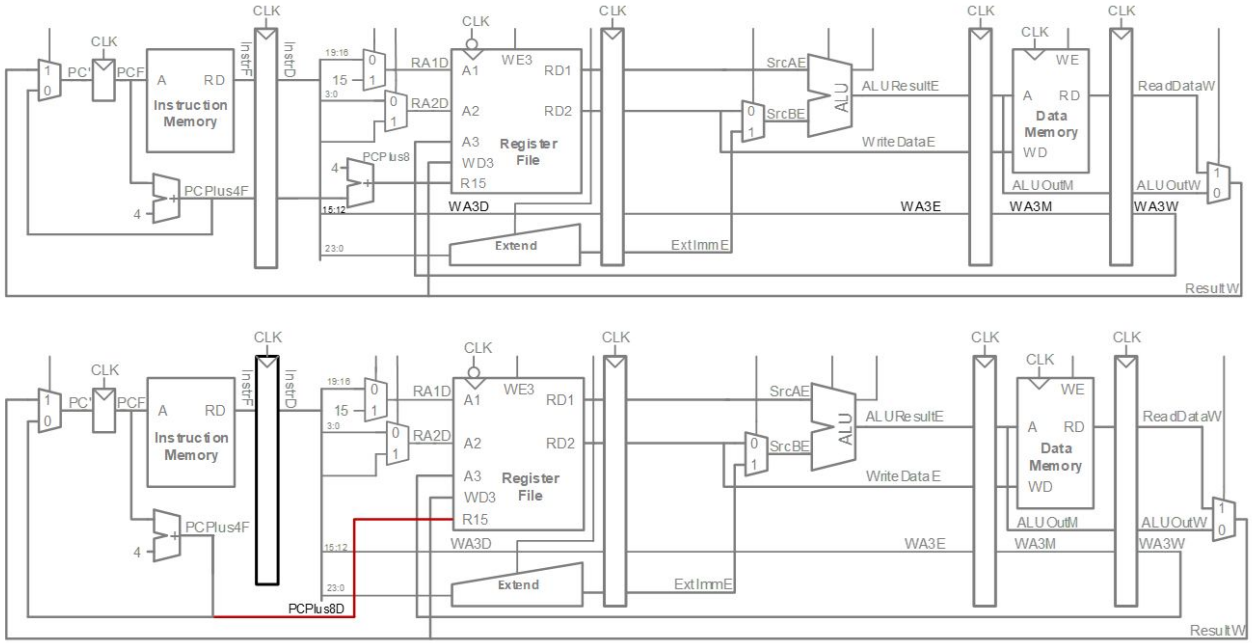
Corrected Pipelined Datapath



- *WA3* must arrive at same time as *Result*
- Register file written on falling edge of *CLK*

Optimized Pipelined Datapath

Remove adder by using PCPlus4F after PC has been updated to PC+4

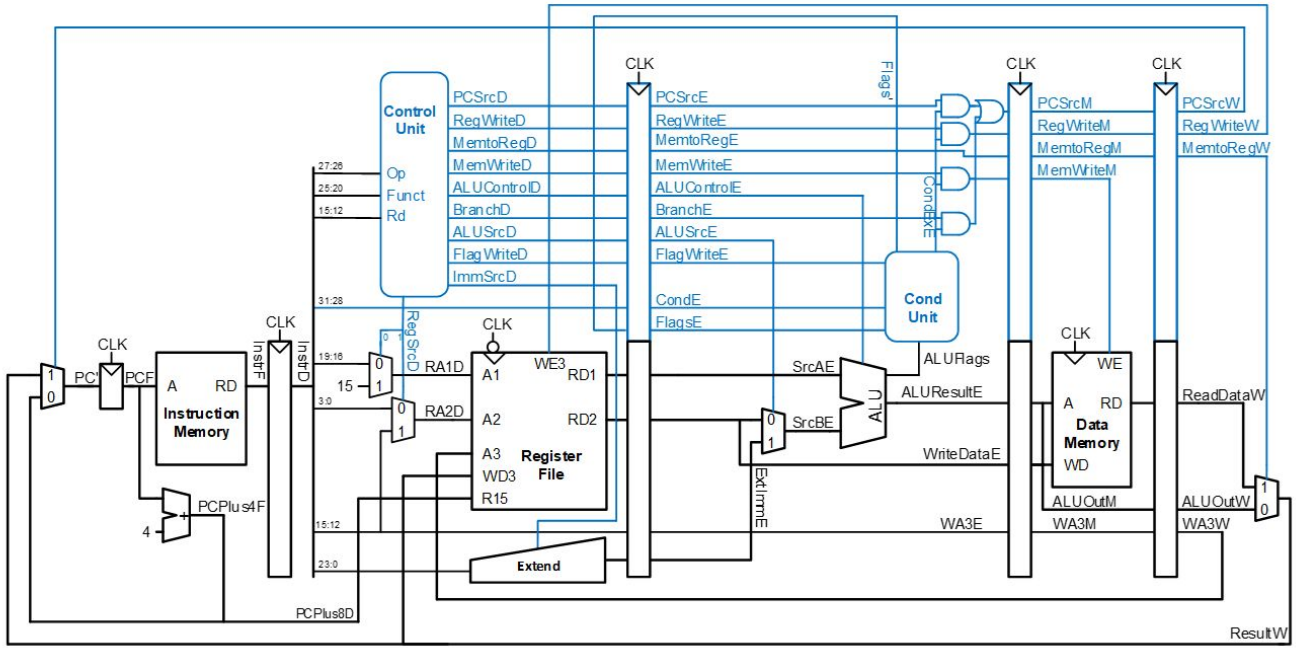


Rahul Bhadani

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Pipelined Processor Control

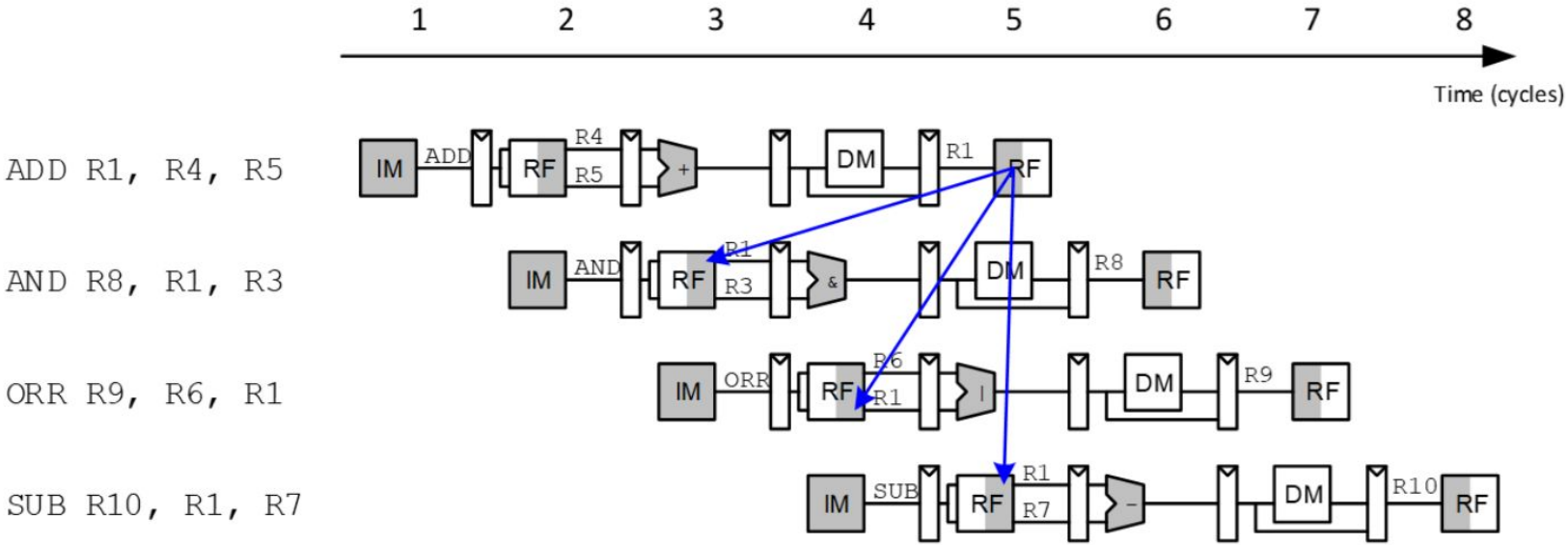
- Same control unit as single-cycle processor
- Control delayed to proper pipeline stage



Pipeline Hazards

- When an instruction depends on result from instruction that hasn't completed
- Types:
  - **Data hazard:** register value not yet written back to register file
  - **Control hazard:** next instruction not decided yet (caused by branch)

Data Hazard

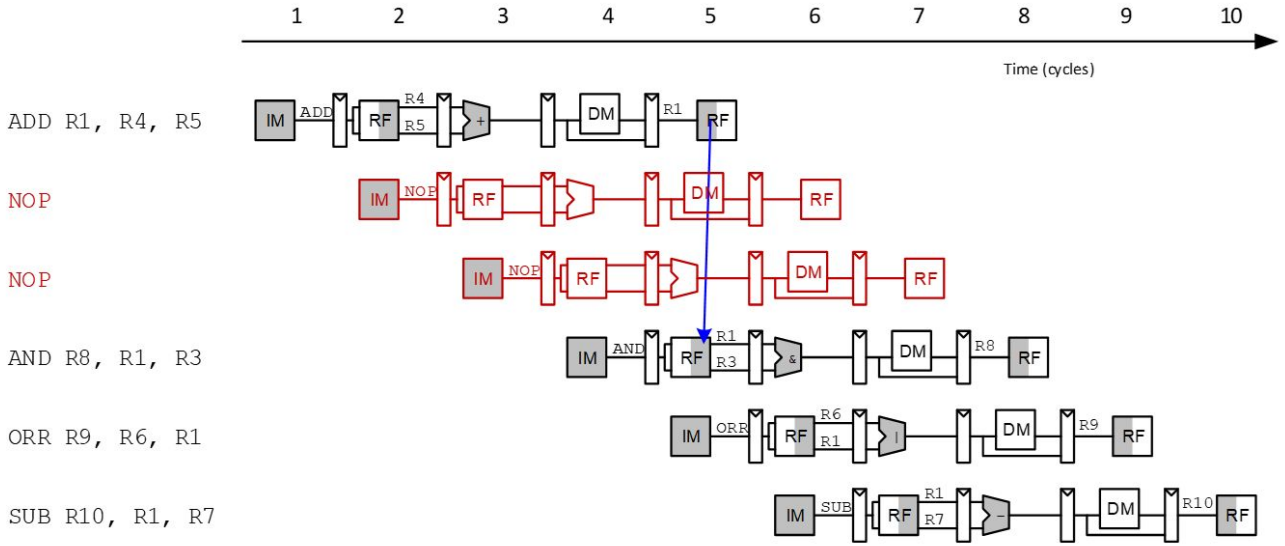

1
2
3
4
5
6
7
8
Time (cycles)
ADD R1, R4, R5
AND R8, R1, R3
ORR R9, R6, R1
SUB R10, R1, R7
IM
ADD
RF
R4
R5
+
DM
R1
RF
IM
AND
RF
R1
R3
&
DM
R8
RF
IM
ORR
RF
R6
R1
|
DM
R9
RF
IM
SUB
RF
R1
R7
-
DM
R10
RF

Handling Data Hazards

- Insert NOPs in code at compile time
- Rearrange code at compile time
- Forward data at run time
- Stall the processor at run time

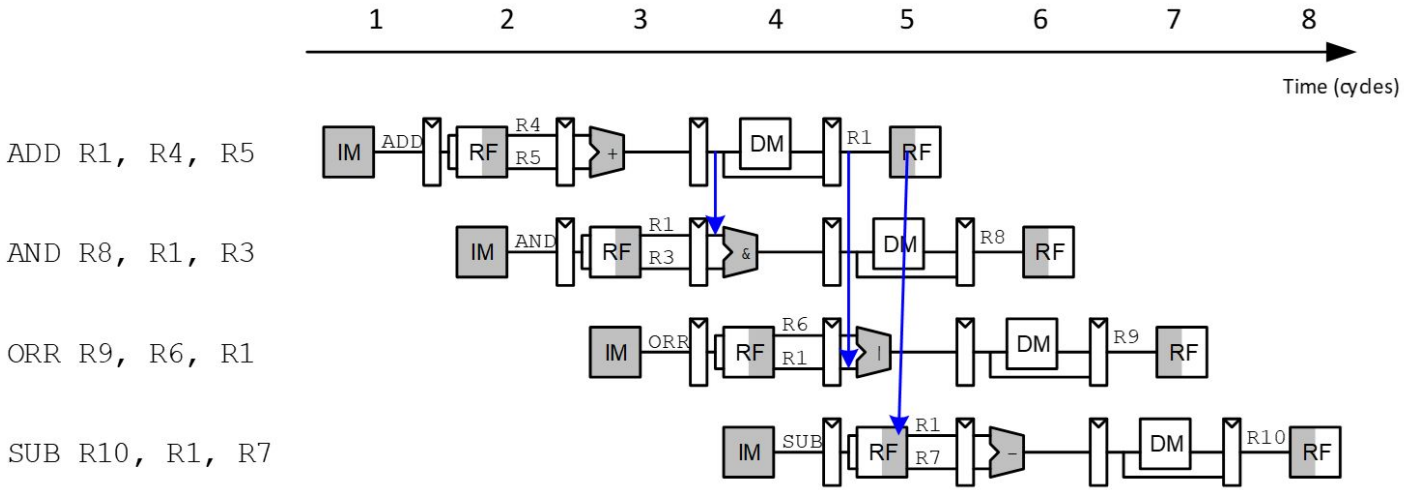
Compile-Time Hazard Elimination

1. Insert enough NOPs for result to be ready
2. Or move independent useful instructions forward

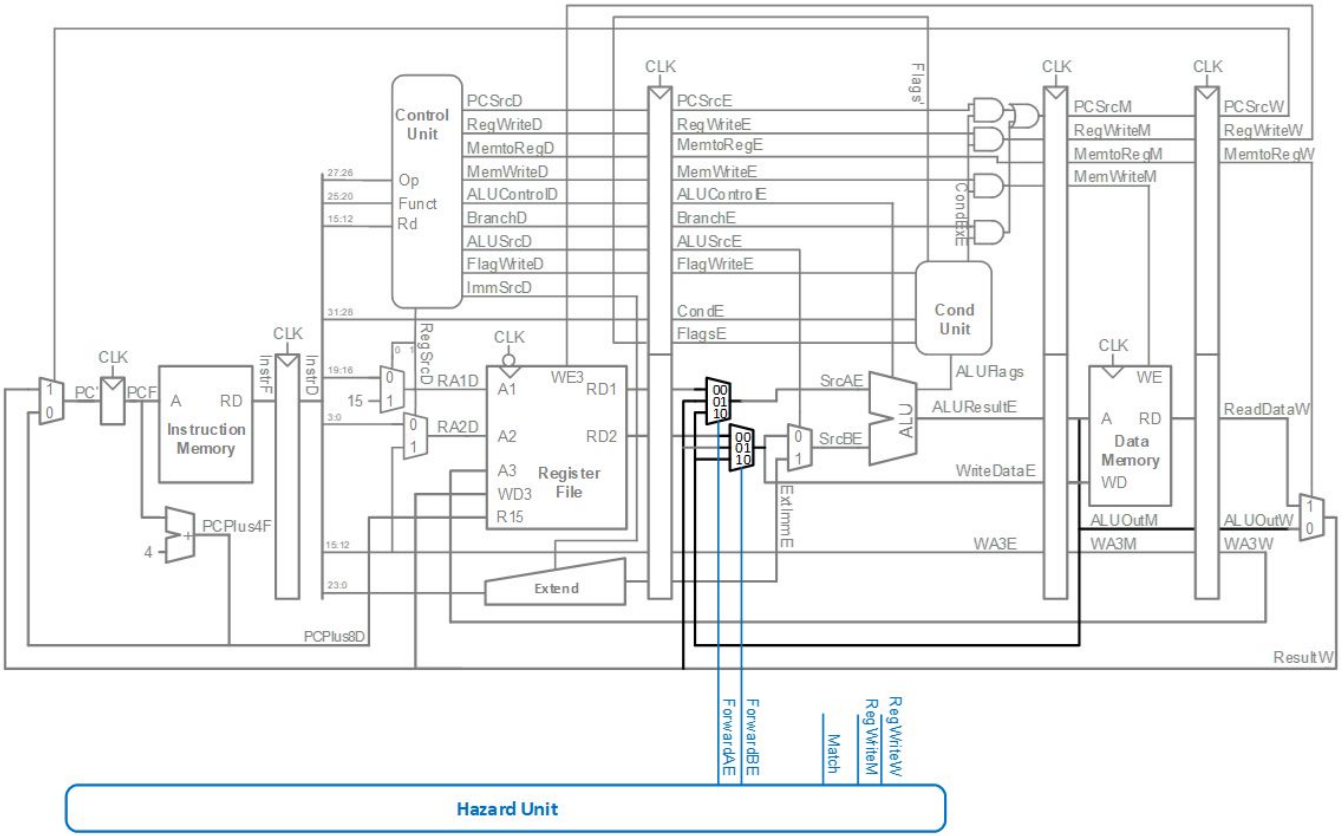


Forward Data at Run Time

1. Check if register read in Execute stage matches register written in Memory or Writeback stage
2. If so, forward the result



Data Forwarding



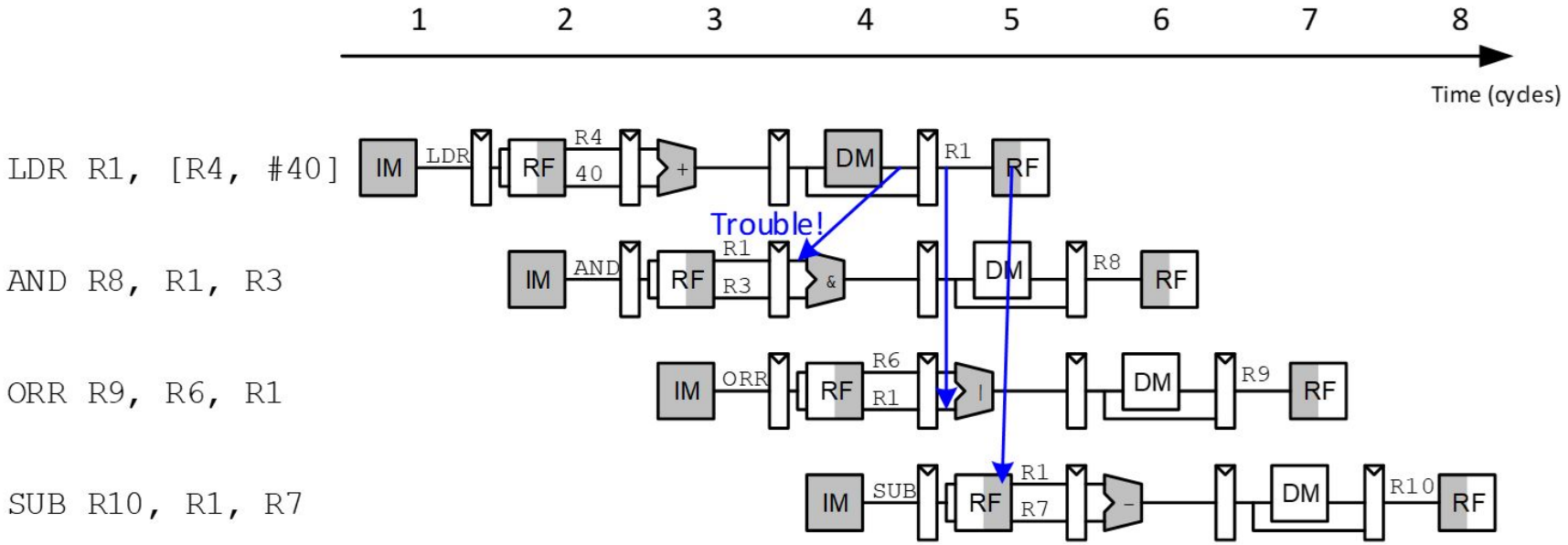
Data Forwarding

- **Execute** stage register matches **Memory** stage register?
  Match_1E_M = (RA1E == WA3M)
  Match_2E_M = (RA2E == WA3M)
- **Execute** stage register matches **Writeback** stage register?
  Match_1E_W = (RA1E == WA3W)
  Match_2E_W = (RA2E == WA3W)
- If it matches, forward result:

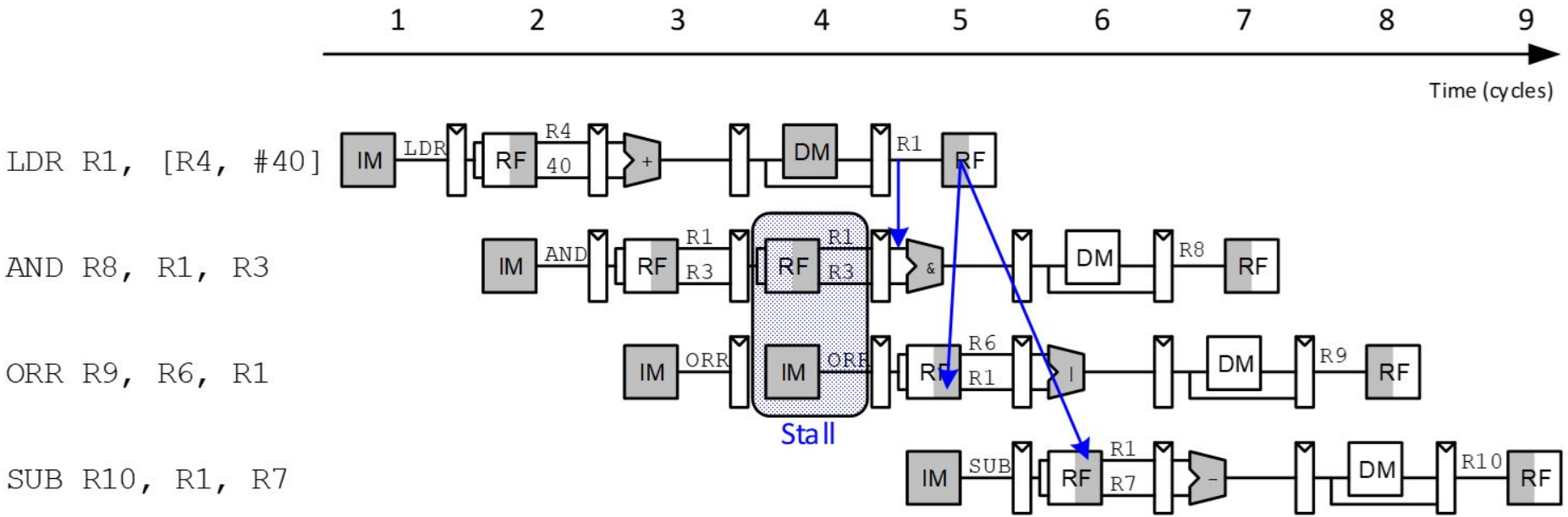
```
if      (Match_1E_M • RegWriteM)  ForwardAE = 10;
else if (Match_1E_W • RegWriteW) ForwardAE = 01;
else                             ForwardAE = 00;
```

ForwardBE same but with Match2E

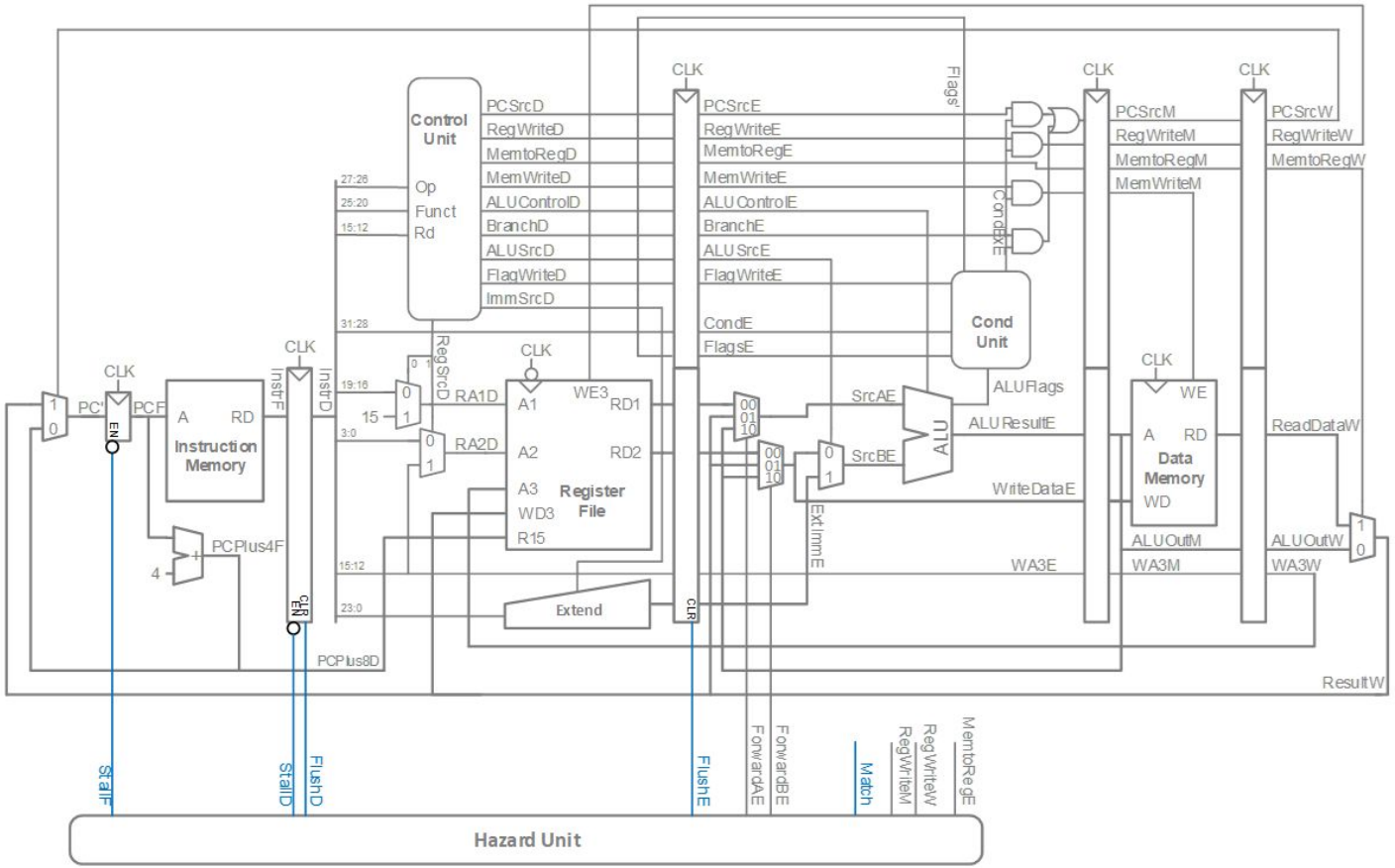
Stalling


1
2
3
4
5
6
7
8
Time (cycles)
LDR R1, [R4, #40]
AND R8, R1, R3
ORR R9, R6, R1
SUB R10, R1, R7
IM
LDR
RF
R4
40
+
DM
R1
RF
Trouble!
IM
AND
RF
R1
R3
&
DM
R8
RF
IM
ORR
RF
R6
R1
|
DM
R9
RF
IM
SUB
RF
R1
R7
-
DM
R10
RF

Stalling


1
2
3
4
5
6
7
8
9
Time (cycles)
LDR R1, [R4, #40]
AND R8, R1, R3
ORR R9, R6, R1
SUB R10, R1, R7
IM
LDR
RF
R4
40
+
DM
R1
RF
IM
AND
RF
R1
R3
RF
R1
R3
&
DM
R8
RF
IM
ORR
IM
ORR
RF
R6
R1
|
DM
R9
RF
Stall
IM
SUB
RF
R1
R7
-
DM
R10
RF

Stalling Hardware



Stalling Logic

- Is either source register in the Decode stage the same as the one being written in the Execute stage?

$$Match\_12D\_E = (RA1D == WA3E) + (RA2D == WA3E)$$

- Is a LDR in the Execute stage AND $Match\_12D\_E$?

$$ldrstall = Match\_12D\_E \cdot MemtoRegE$$
$$StallF = StallD = FlushE = ldrstall$$

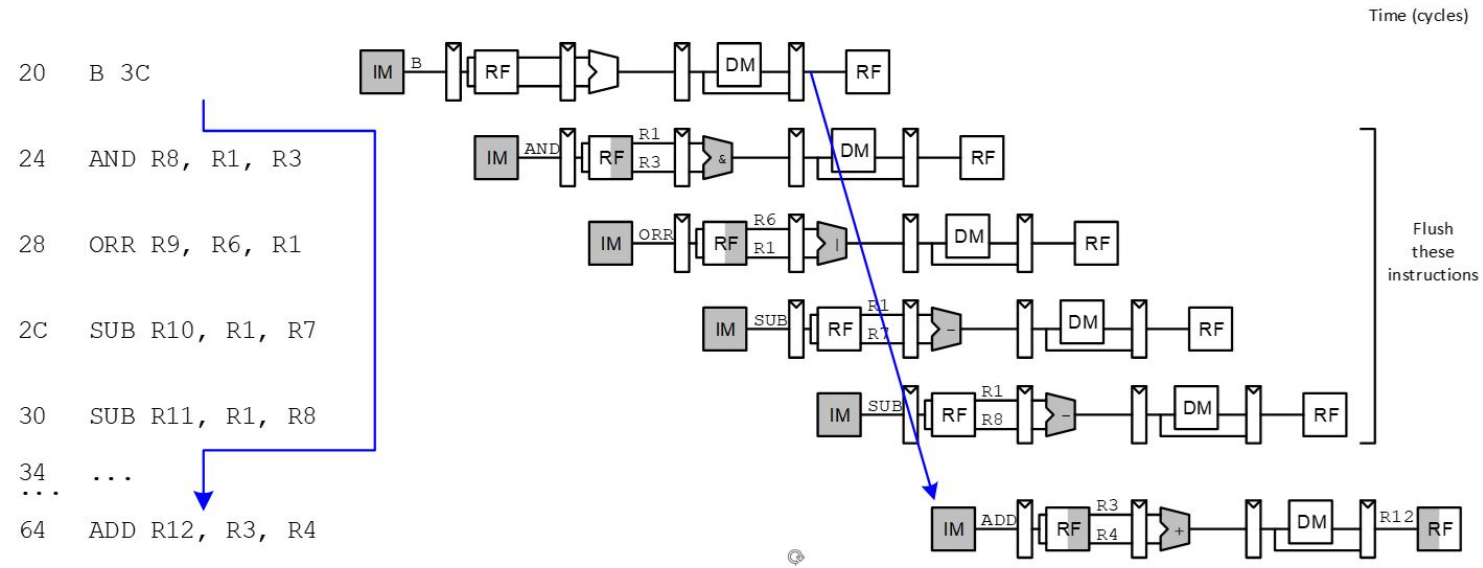
Control Hazards

- B:
  - branch not determined until the Writeback stage of pipeline
  - Instructions after branch fetched before branch occurs
  - These 4 instructions must be flushed if branch happens
- **Writes to PC (R15) similar**

Control Hazards

Branch misprediction penalty

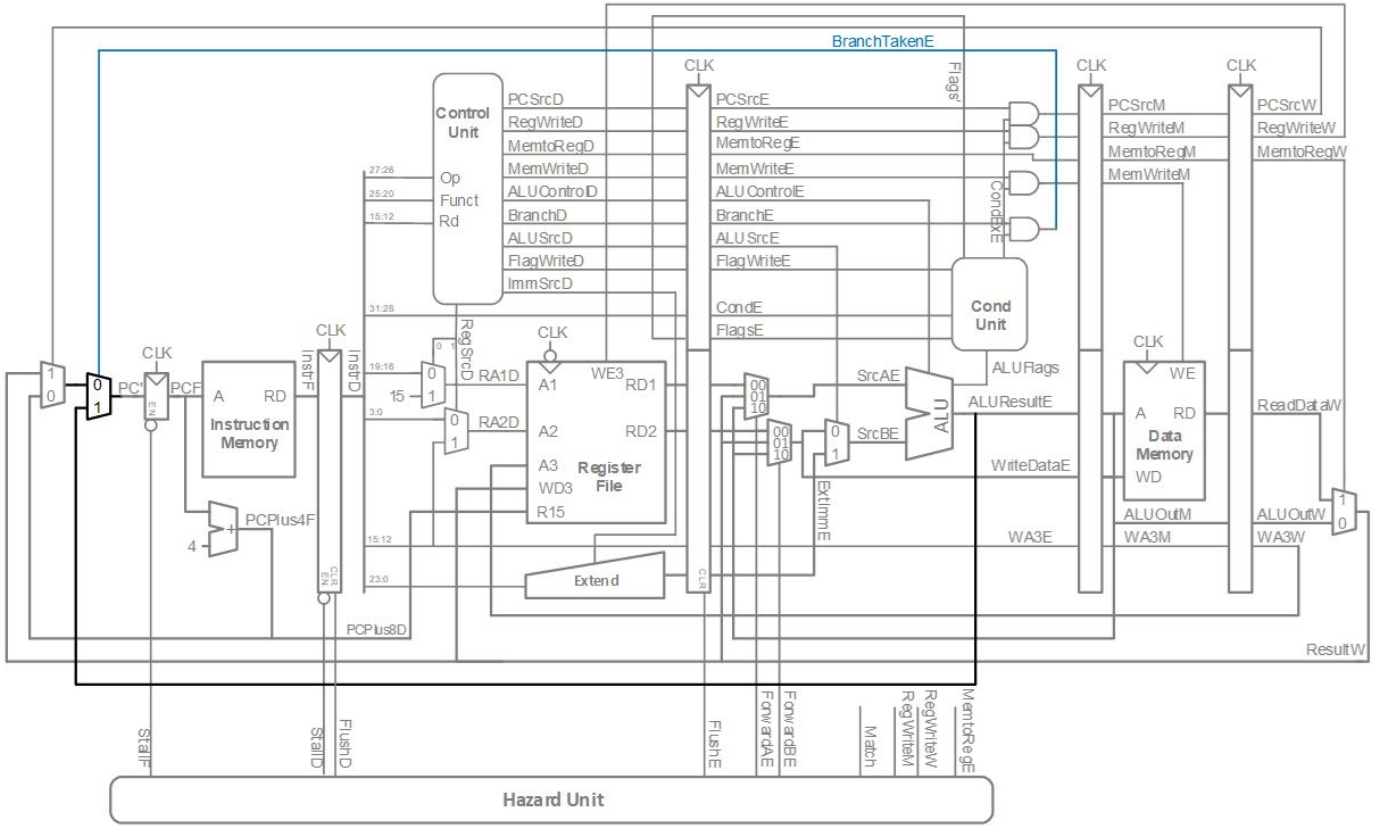
- number of instruction flushed when branch is taken (4)
- May be reduced by determining BTA earlier



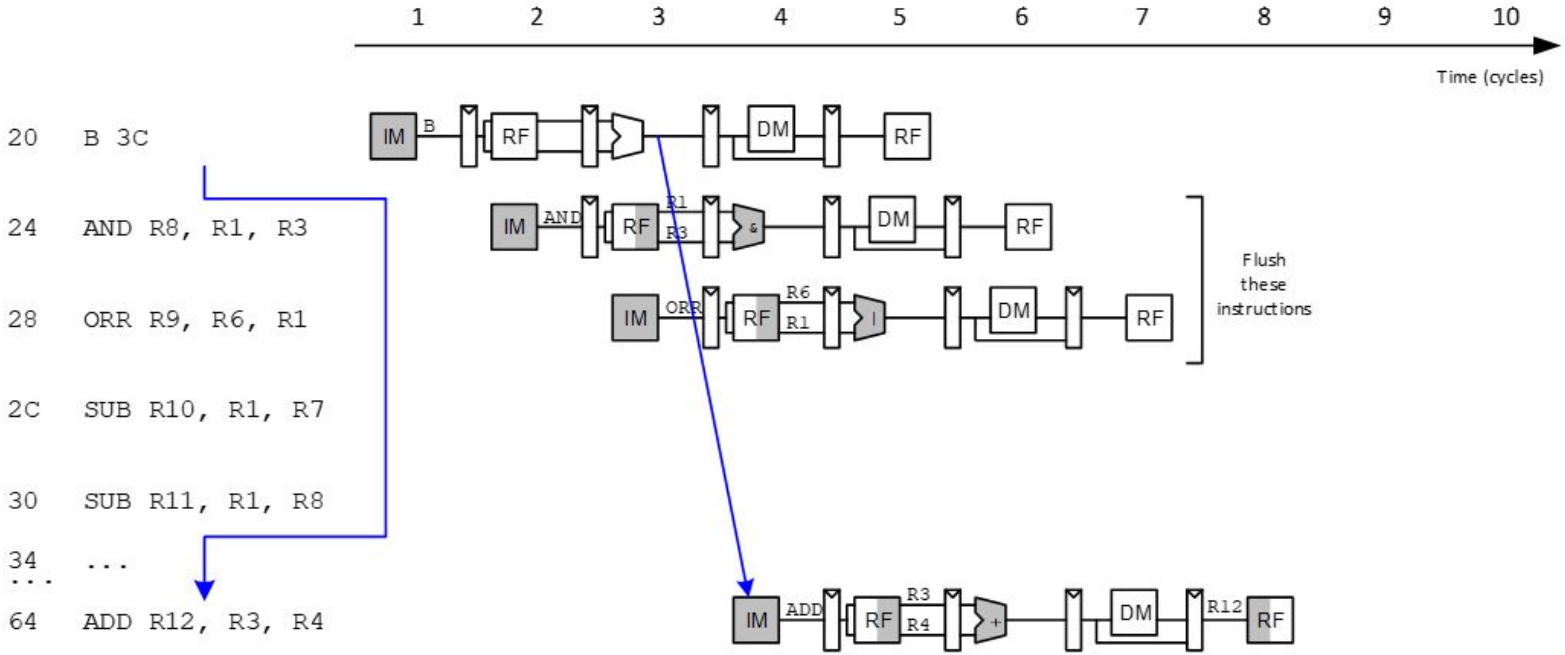
Early Branch Resolution

- **Determine BTA in Execute stage**
  - Branch misprediction penalty = 2 cycles
- **Hardware changes**
  - Add a branch multiplexer before *PC* register to select BTA from *ALUResultE*
  - Add *BranchTakenE* select signal for this multiplexer (only asserted if branch condition satisfied)
  - *PCSrcW* now only asserted for writes to *PC*

Pipelined processor with Early BTA



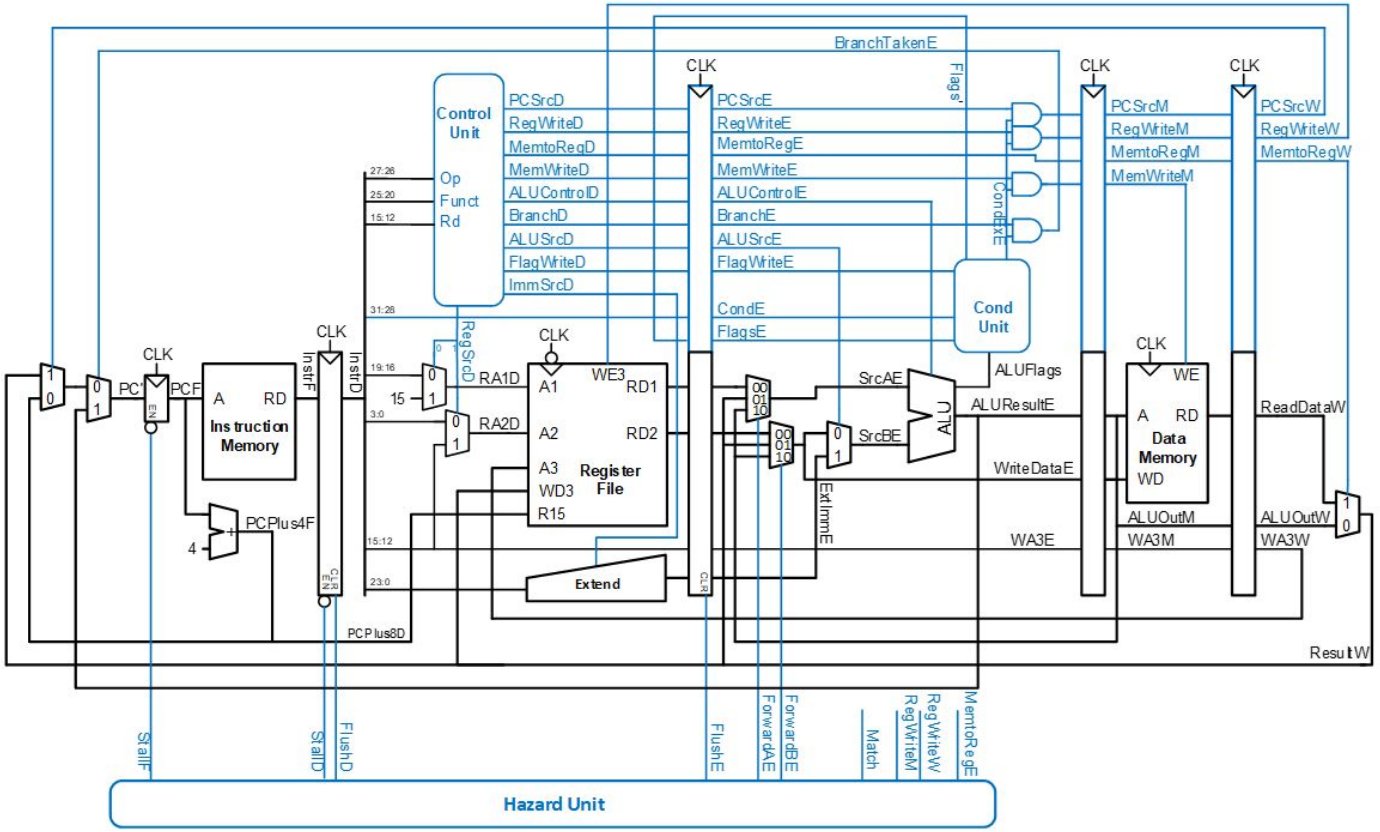
Control Hazards with Early BTA



Control Stalling Logic

- ***PCWrPendingF* = 1** if write to *PC* in Decode, Execute or Memory
  PCWrPendingF = PCSrcD + PCSrcE + PCSrcM
- **Stall Fetch** if *PCWrPendingF*
  StallF = ldrStallD + PCWrPendingF
- **Flush Decode** if *PCWrPendingF* OR *PC* is written in Writeback OR branch is taken
  FlushD = PCWrPendingF + PCSrcW + BranchTakenE
- **Flush Execute** if branch is taken
  FlushE = ldrStallD + BranchTakenE
- **Stall Decode** if *ldrStallD* (as before)
  StallD = ldrStallD

ARM Pipelined Processor with Hazard Unit



Pipelined Performance Example

- **SPECINT2000 benchmark:**
  - 25% loads
  - 10% stores
  - 13% branches
  - 52% R-type
- **Suppose:**
  - 40% of loads used by next instruction
  - 50% of branches mispredicted
- **What is the average CPI?**

Pipelined Performance Example

- **What is the average CPI?**
  - Load CPI = 1 when not stalling, 2 when stalling

    So, $\mathbf{CPI_{lw}} = 1(0.6) + 2(0.4) = 1.4$
  - Branch CPI = 1 when not stalling, 3 when stalling

    So, $\mathbf{CPI_{beq}} = 1(0.5) + 3(0.5) = 2$

$$\textbf{Average CPI} = (0.25)(1.4) + (0.1)(1) + (0.13)(2) + (0.52)(1) = \mathbf{1.23}$$

Pipelined Performance

Pipelined processor critical path:

$$T_{c3} = \max \left[\begin{array}{ll} t_{pcq} + t_{mem} + t_{setup} & \text{Fetch} \\ 2(t_{RFread} + t_{setup}) & \text{Decode} \\ t_{pcq} + 2t_{mux} + t_{ALU} + t_{setup} & \text{Execute} \\ t_{pcq} + t_{mem} + t_{setup} & \text{Memory} \\ 2(t_{pcq} + t_{mux} + t_{RFwrite}) \end{array} \right] \text{Writeback}$$

Cycle time: $T_{c3} = ?$

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60
Register file write	$t_{RFwrite}$	70

Pipelined Performance

Pipelined processor critical path:

$$T_{c3} = \max \begin{cases} t_{pcq} + t_{mem} + t_{setup} & \text{Fetch} \\ 2(t_{RFread} + t_{setup}) & \text{Decode} \\ t_{pcq} + 2t_{mux} + t_{ALU} + t_{setup} & \text{Execute} \\ t_{pcq} + t_{mem} + t_{setup} & \text{Memory} \\ 2(t_{pcq} + t_{mux} + t_{RFwrite}) & \text{Writeback} \end{cases}$$

Cycle time: $T_{c3} = = 2(t_{RFread} + t_{setup})$

$= 2[100 + 50]$ ps $=$ **300 ps**

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60
Register file write	$t_{RFwrite}$	70

Program with 100 billion instructions

Execution Time = (# instructions) × CPI × T_c

= (100 × 109)(1.23)(300 × 10-12) = 36.9 seconds

Processor Performance Comparison

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
Single-cycle	84	1
Multicycle	140	0.6
Pipelined	36.9	2.28

Advanced Microarchitecture

- Deep Pipelining
- Micro-operations
- Branch Prediction
- Superscalar Processors
- Out of Order Processors
- Register Renaming
- SIMD
- Multithreading
- Multiprocessors

Deep Pipelining

- 10-20 stages typical
- Number of stages limited by:
  - Pipeline hazards
  - Sequencing overhead
  - Power
  - Cost

Micro-operations

- Decompose more complex instructions into a series of simple instructions called *micro-operations* (*micro-ops* or *μ-ops*)
- At run-time, complex instructions are decoded into one or more micro-ops
- Used heavily in CISC (complex instruction set computer) architectures (e.g., x86)
- Used for some ARM instructions, for example:

Complex Op Micro-op Sequence

LDR R1, [R2], #4 LDR R1, [R2]

ADD R2, R2, #4

Without *μ*-ops, would need 2nd write port on the register file

Micro-operations …

- Allow for dense code (fewer memory accesses)
- Yet preserve simplicity of RISC hardware
- ARM strikes balance by choosing instructions that:
  - Give better code density than pure RISC instruction sets (such as MIPS)
  - Enable more efficient decoding than CISC instruction sets (such as x86)

Branch Prediction

- Guess whether branch will be taken
  - Backward branches are usually taken (loops)
  - Consider history to improve guess
- Good prediction reduces fraction of branches requiring a flush

Branch Prediction

- Ideal pipelined processor: CPI = 1
- Branch misprediction increases CPI
- **Static branch prediction:**
  - Check direction of branch (forward or backward)
  - If backward, predict taken
  - Else, predict not taken
- **Dynamic branch prediction:**
  - Keep history of last several hundred (or thousand) branches in *branch target buffer*, record:
    - Branch destination
    - Whether branch was taken

Branch Prediction Example

ARM Assembly Code

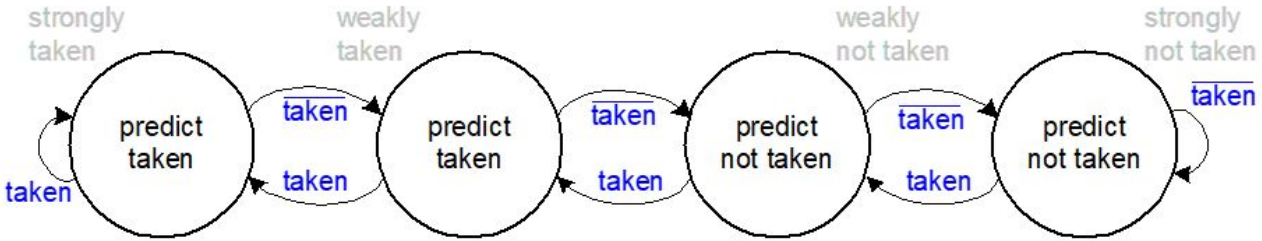
```
  MOV R1, #0 ; R1 = sum
  MOV R0, #0 ; R0 = i

FOR ; for (i=0; i<10; i=i+1)
  CMP R0, #10
  BGE DONE
  ADD R1, R1, R0 ; sum = sum + i
  ADD R0, R0, #1
  B   FOR

DONE
```

1-Bit Branch Predictor

- Remembers whether branch was taken the last time and does the same thing
- Mispredicts first and last branch of loop

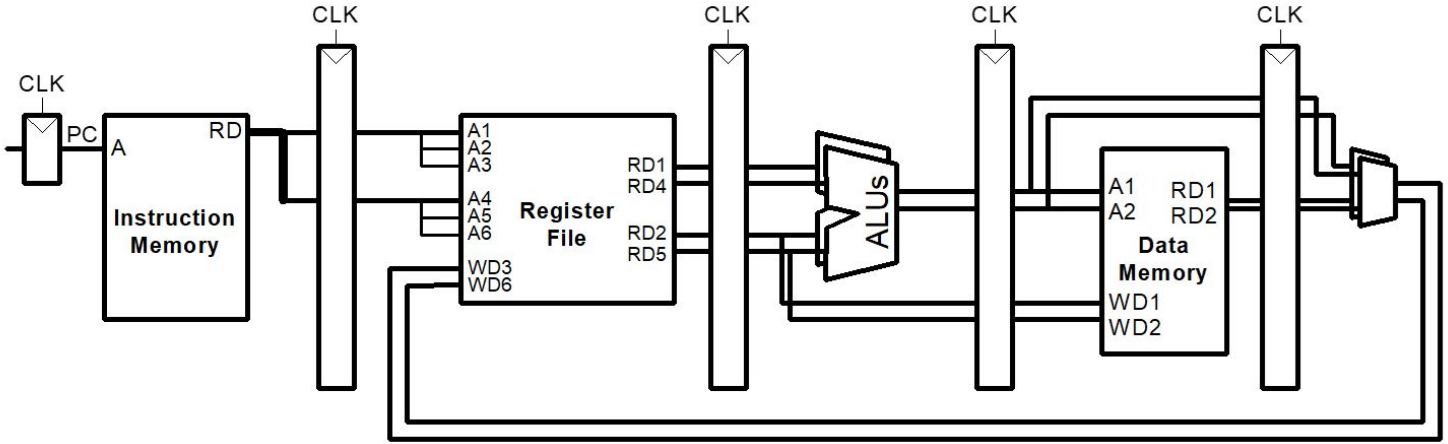
2-Bit Branch Predictor

Only mispredicts last branch of loop



Superscalar

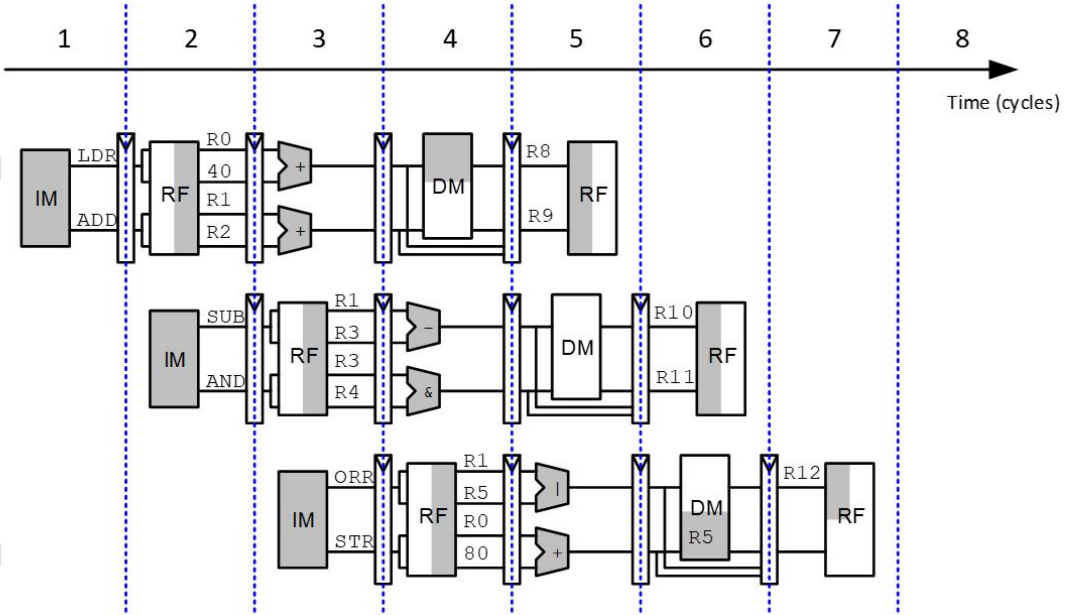
- Multiple copies of datapath execute multiple instructions at once
- Dependencies make it tricky to issue multiple instructions at once



Superscalar Example

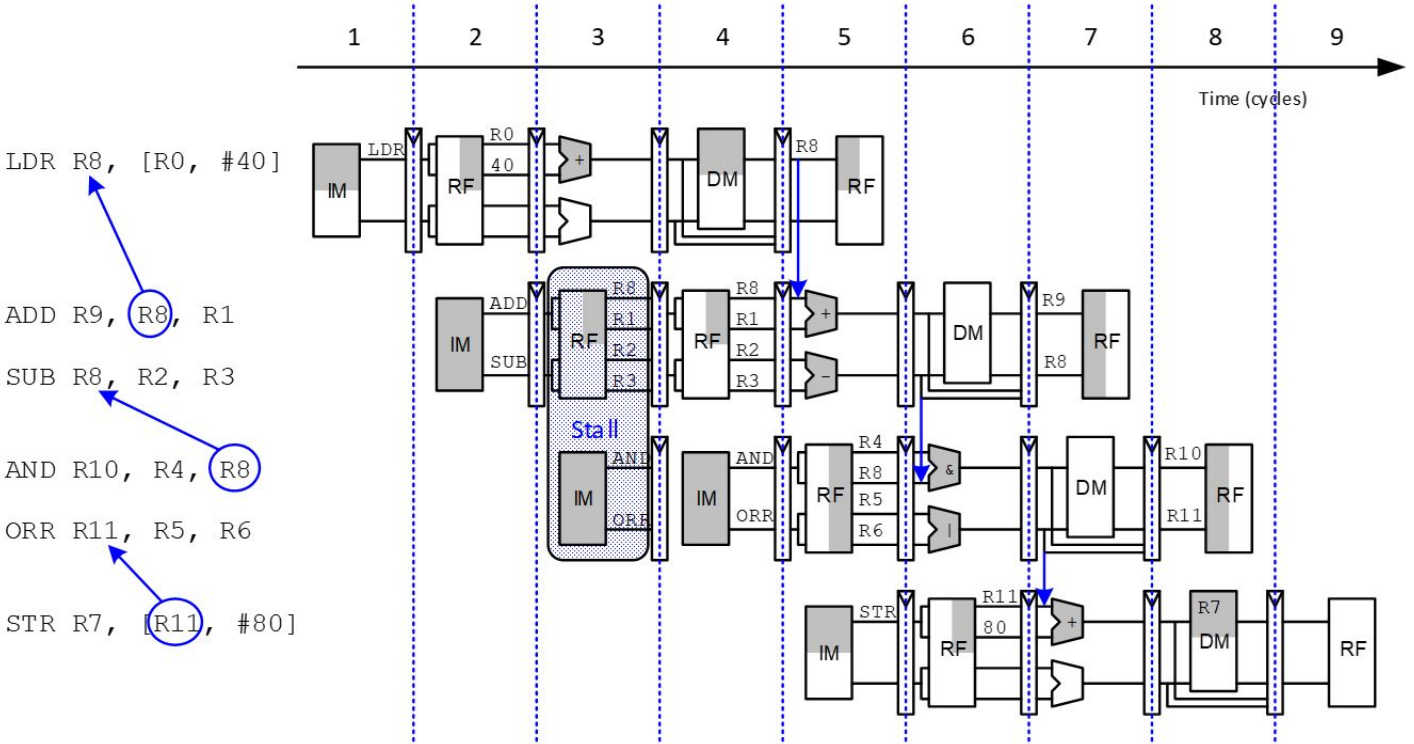
Ideal IPC: 2
Actual IPC: 2

```
LDR R8, [R0, #40]
ADD R9, R1, R2

SUB R10, R1, R3
AND R11, R3, R4

ORR R12, R1, R5
STR R5, [R0, #80]
```



Superscalar with Dependencies

Ideal IPC: 2
Actual IPC: 6/5 = 1.2



Out of Order Processor

- Looks ahead across multiple instructions
- Issues as many instructions as possible at once
- Issues instructions out of order (as long as no dependencies)
- **Dependencies:**
  - **RAW** (read after write): one instruction writes, later instruction reads a register
  - **WAR** (write after read): one instruction reads, later instruction writes a register
  - **WAW** (write after write): one instruction writes, later instruction writes a register

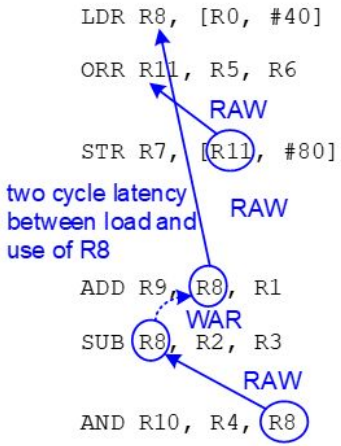
Out of Order Processor

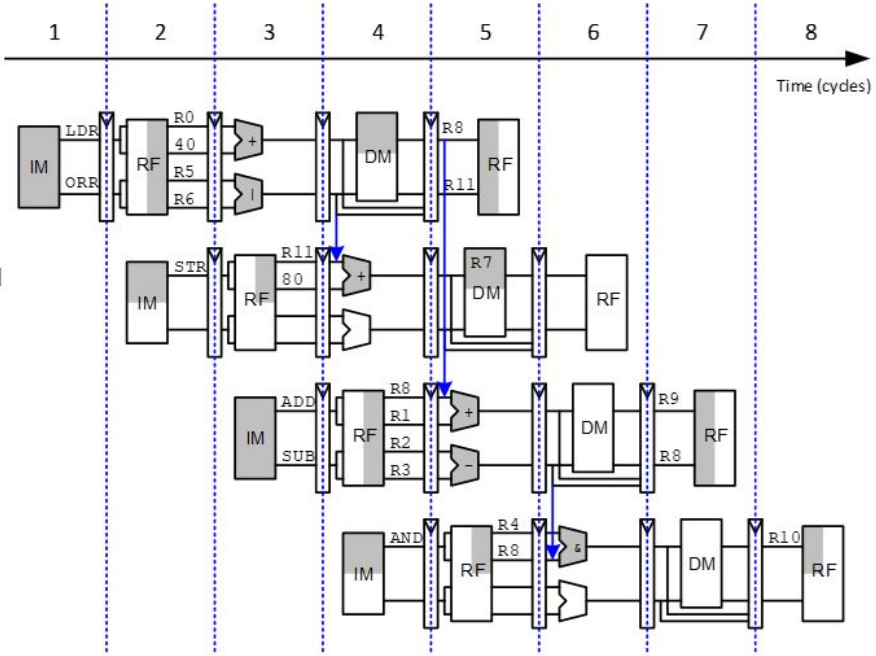
- **Instruction level parallelism (ILP):** number of instruction that can be issued simultaneously (average < 3)
- **Scoreboard:** table that keeps track of:
  - Instructions waiting to issue
  - Available functional units
  - Dependencies

Out of Order Processor Example

ARM Assembly Code

```
LDR  R8, [R0, #40]
ADD  R9,  R8, R1
SUB  R8,  R2, R3 Ideal IPC: 2
AND  R10, R4, R8 Actual IPC: 6/4 = 1.5
ORR  R11, R5, R6
STR  R7, [R11, #80]
```

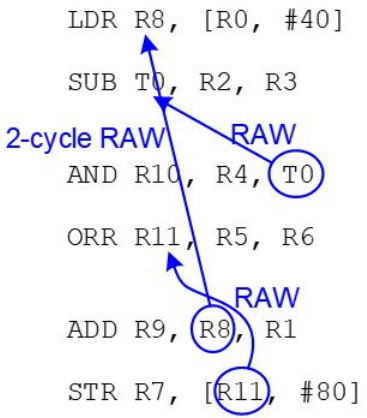


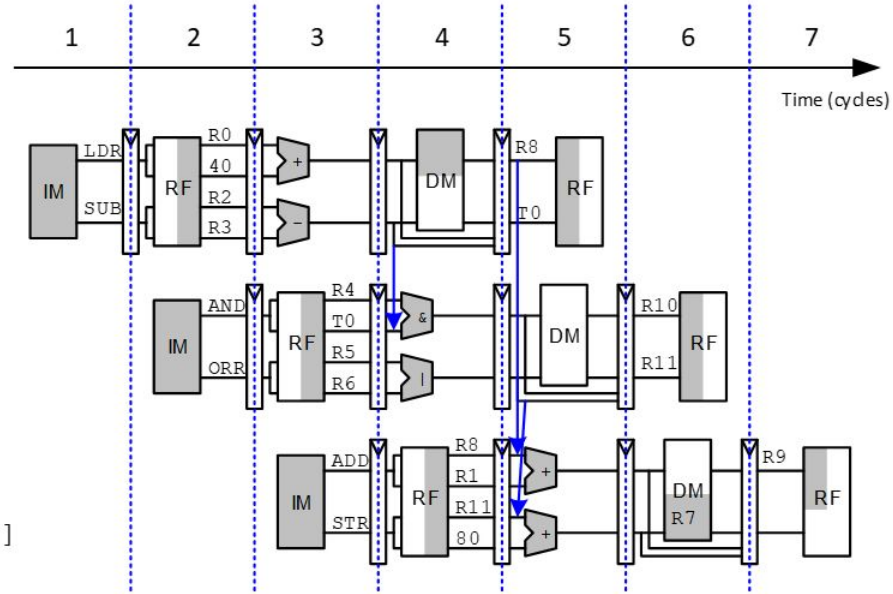


Register Renaming

ARM Assembly Code

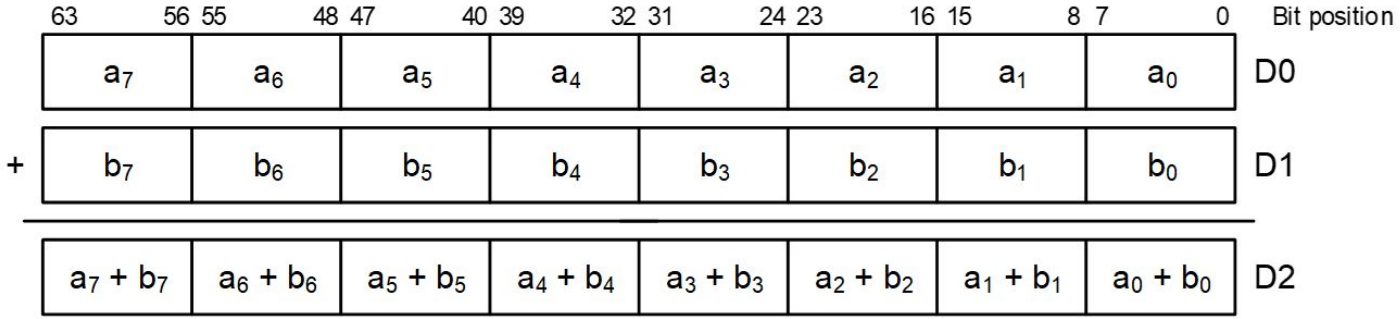
```
LDR  R8, [R0, #40]
ADD  R9,  R8, R1
SUB  R8,  R2, R3 Ideal IPC: 2
AND  R10, R4, R8 Actual IPC: 6/3 = 2
ORR  R11, R5, R6
STR  R7, [R11, #80]
```





SIMD

- Single Instruction Multiple Data (SIMD)
  - Single instruction acts on multiple pieces of data at once
  - Common application: graphics
  - Perform short arithmetic operations (also called *packed arithmetic*)
- For example, add eight 8-bit elements



Advanced Architecture Techniques

- **Multithreading**
  - Wordprocessor: thread for typing, spell checking, printing
- **Multiprocessors**
  - Multiple processors (cores) on a single chip

Threading: Definitions

- **Process:** program running on a computer
  - Multiple processes can run at once: e.g., surfing Web, playing music, writing a paper
- **Thread:** part of a program
  - Each process has multiple threads: e.g., a word processor may have threads for typing, spell checking, printing

Threads in Conventional Processor

- One thread runs at once
- When one thread stalls (for example, waiting for memory):
  - Architectural state of that thread stored
  - Architectural state of waiting thread loaded into processor and it runs
  - Called **context switching**
- Appears to user like all threads running simultaneously

Multithreading

- Multiple copies of architectural state
- Multiple threads **active** at once:
  - When one thread stalls, another runs immediately
  - If one thread can't keep all execution units busy, another thread can use them
- Does not increase instruction-level parallelism (ILP) of single thread, but increases throughput

  Intel calls this "hyperthreading"

Multiprocessors

- Multiple processors (cores) with a method of communication between them
- Types:
  - **Homogeneous:** multiple cores with shared main memory
  - **Heterogeneous:** separate cores for different tasks (for example, DSP and CPU in cell phone)
  - **Clusters:** each core has own memory system

Additional Readings

- Patterson & Hennessy's: *Computer Architecture: A Quantitative Approach*
- Conferences:
  - ISCA (International Symposium on Computer Architecture)
  - HPCA (International Symposium on High Performance Computer Architecture)