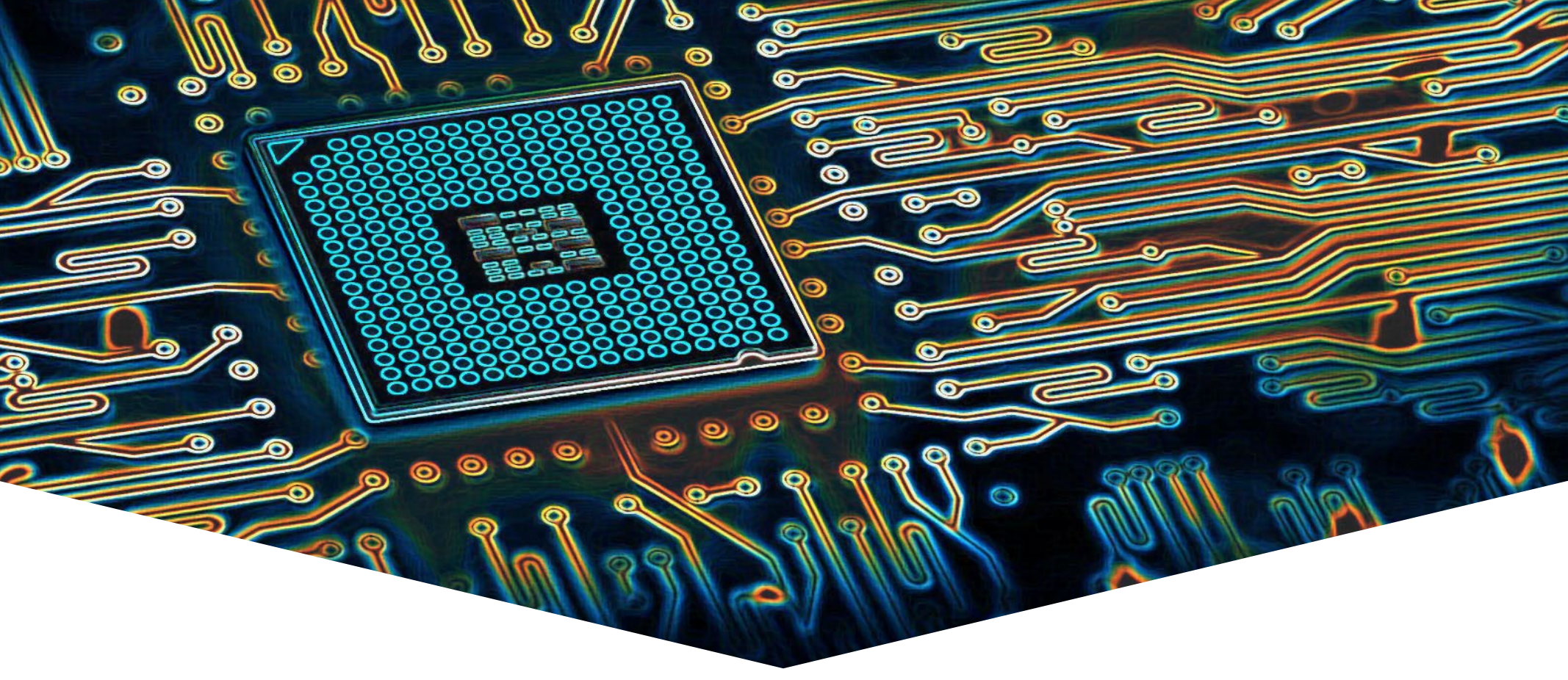


RTS10 Week 2

Leerdoelen Week 2 Theoriedeel

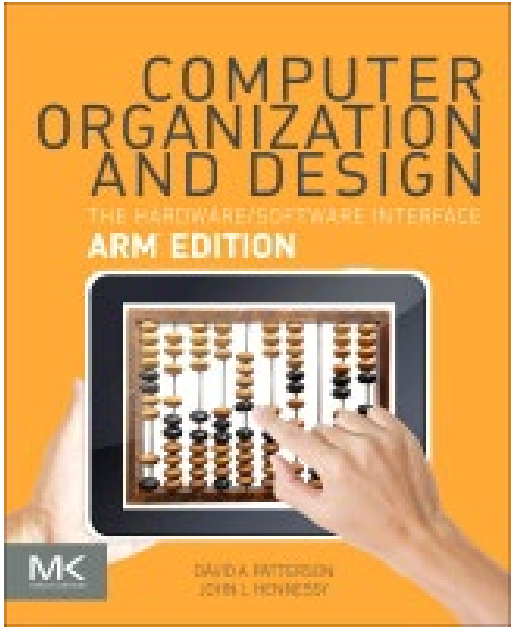
Leerdoelen week 2 theoriedeel. Je leert:

- hoe een vereenvoudigde versie van de ARMv7-M architectuur, de LEGv7-M architectuur, in hardware geïmplementeerd kan worden;
- deze implementatie versnelt kan worden door het toepassen van:
  - een pipeline;
  - branch prediction;
  - cache geheugen.

Source Reference

The material in this presentation is inspired by chapter 4 of the book:

- Computer Organization and Design
  - The Hardware / Software Interface
    - ARM Edition
    - David A. Patterson, John L. Hennessy
    - ISBN: 9780128017333



Main differences:

- The book uses ARMv8 Cortex-A , I use ARMv7 Cortex-M
- The book uses ARM instructions, I use Thumb instructions

Agenda

- A one cycle implementation of LEGv7-M
- A pipelined implementation of LEGv7-M
- Branch prediction
- Cache memory

Agenda

- A one cycle implementation of LEGv7-M
- A pipelined implementation of LEGv7-M
- Branch prediction
- Cache memory

Introduction

CPU performance factors

- Instruction count
  - Determined by Instruction Set Architecture (ISA) and compiler
- Clocks per instruction (CPI) and Cycle time
  - Determined by CPU hardware

We will examine two LEGv7-M implementations

- A simplified version
- A more realistic pipelined version

Simple subset of Pinky instruction set, shows most aspects

- Memory reference: LDR, STR
- Arithmetic/logical: ADDS, SUBS, EORS, ANDS, ORRS
- Control transfer: CBZ

Pinky subset to implement

Instructie
ADDS.N <Rd>,<Rn>,<Rm>
SUBS.N <Rd>,<Rn>,<Rm>
ANDS.N <Rdn>,<Rm>
EORS.N <Rdn>,<Rm>
ORRS.N <Rdn>,<Rm>
STR.N <Rt>, [<Rn>,#<imm7>]
LDR.N <Rt>, [<Rn>,#<imm7>]
CBZ.N <Rn>,<label>

R(egister)-type

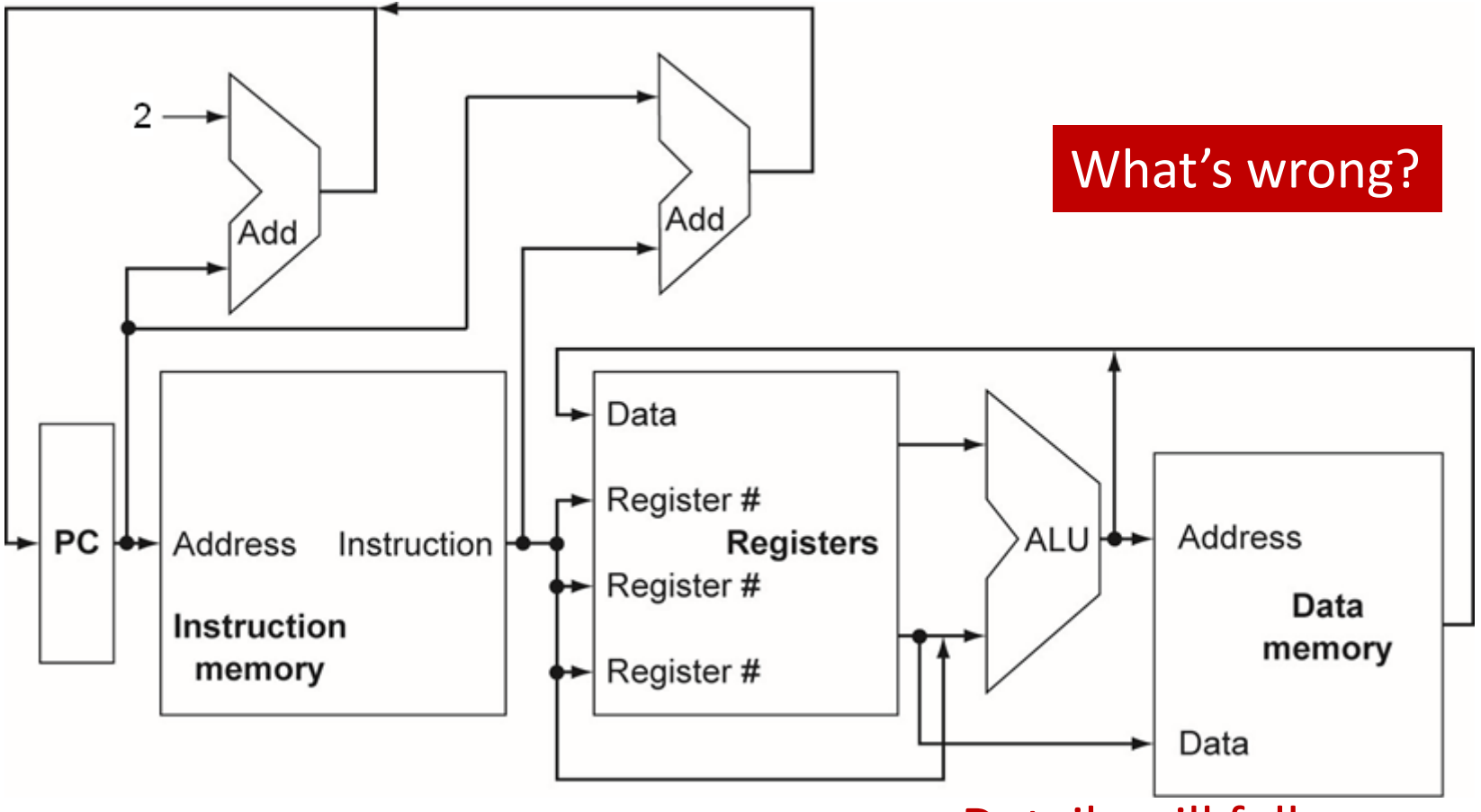
M(emory)-type

C(ompare)B(ranch)-type

Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction type:
  - Use Arithmetic and Logic Unit (ALU) to calculate
    - Arithmetic or logic result for Register-type (ADDS, SUBS, EORS, ANDS, ORRS)
    - Memory address for Memory-type instructions (LDU, STU)
    - Register equals zero for CB-type (CBZ)
  - For CB-type (CBZ) calculate branch target address
  - Access data memory for M-type (LDU, STU)
  - Write to register file for R-type (ADDS, SUBS, EORS, ANDS, ORRS) and load (LDR)
- PC ← PC + 2 or PC ← Branch target address

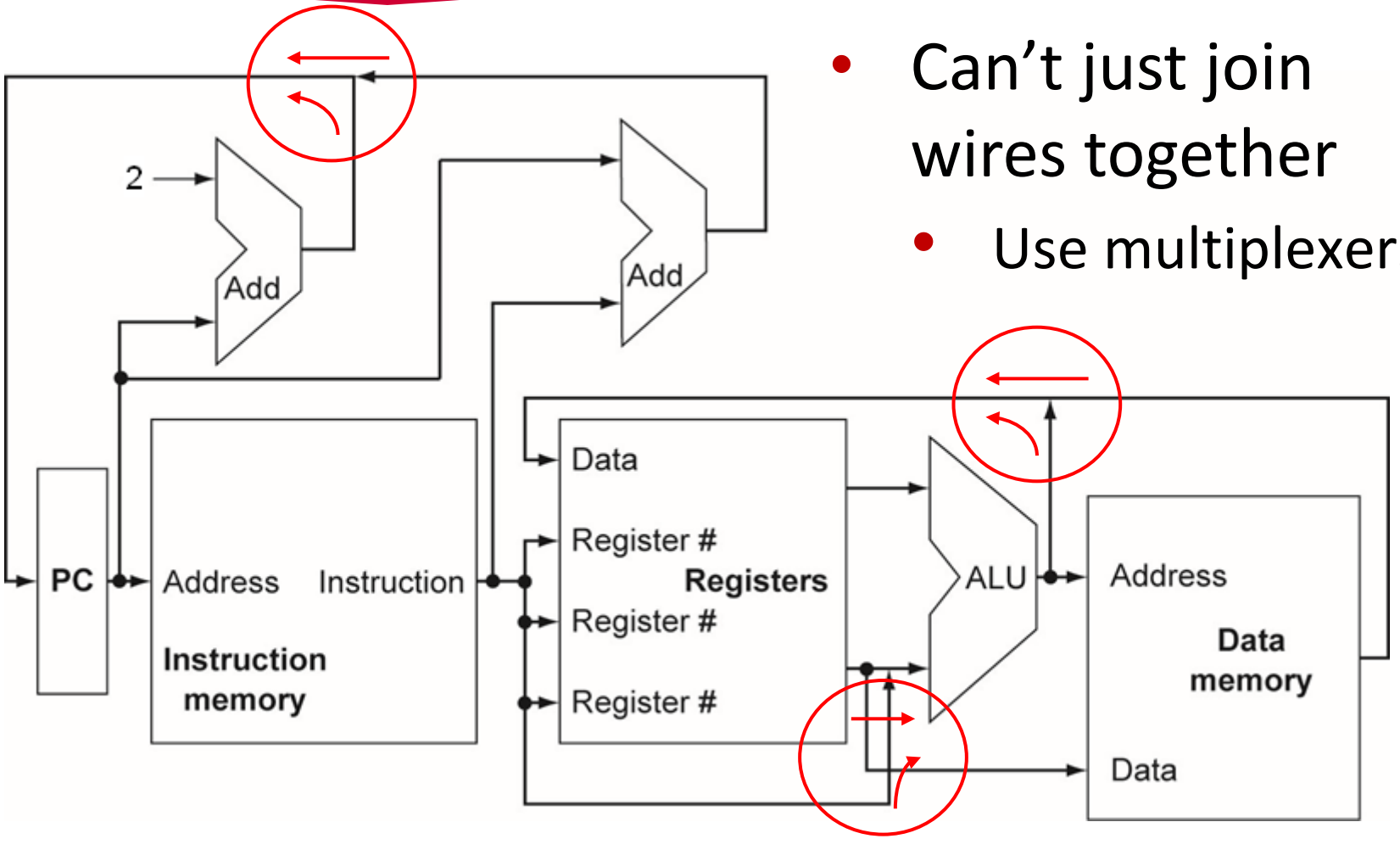
CPU Overview



What's wrong?

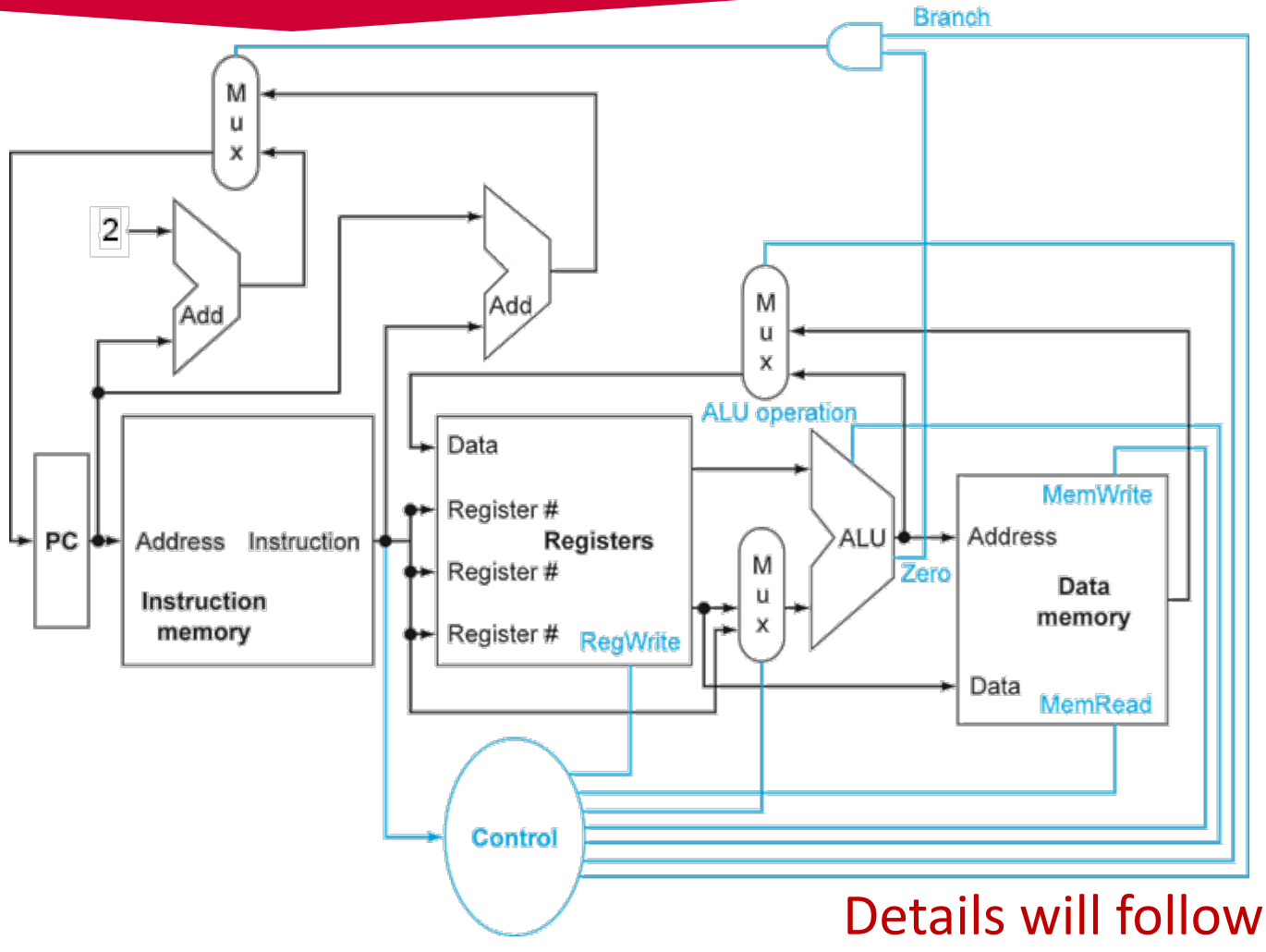
Details will follow…

Can’t join output signals



- Can’t just join wires together
  - Use multiplexers

Control



Details will follow…

Logic Design Basics

>>

Information encoded in binary

- Low voltage = 0, High voltage = 1
- One wire per bit
- Multi-bit data encoded on multi-wire buses

Combinational element

- Operate on data
- Output is a function of input

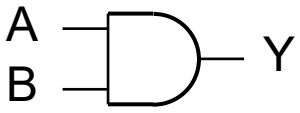
State (sequential) elements

- Store information

Combinational Elements

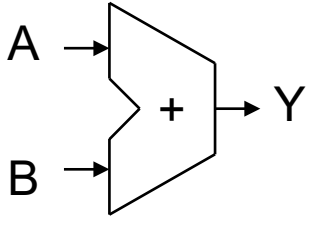
AND gate

Y = A & B



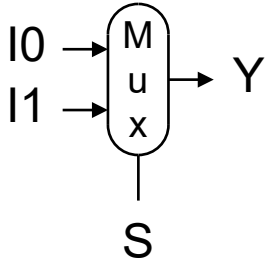
Adder

Y = A + B



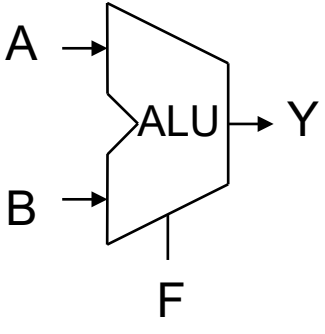
Multiplexer

Y = S ? I1 : I0



Arithmetic/Logic Unit

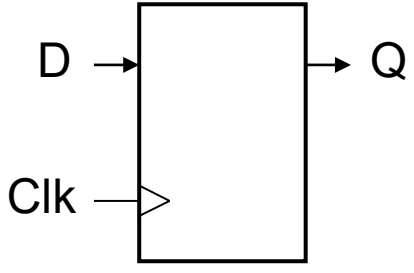
Y = F(A, B)

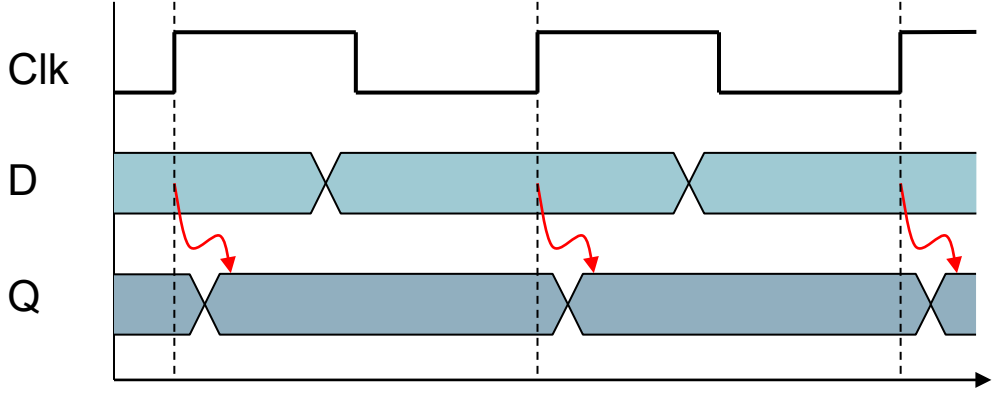


Sequential Elements

Register: stores data in a circuit

- Uses a clock signal to determine when to update the stored value
- Edge-triggered: update when Clk changes from 0 to 1

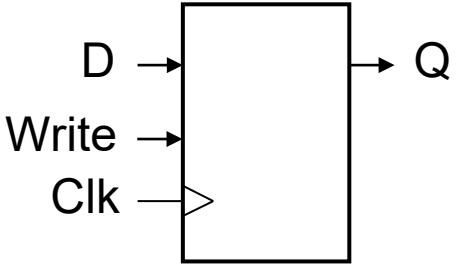


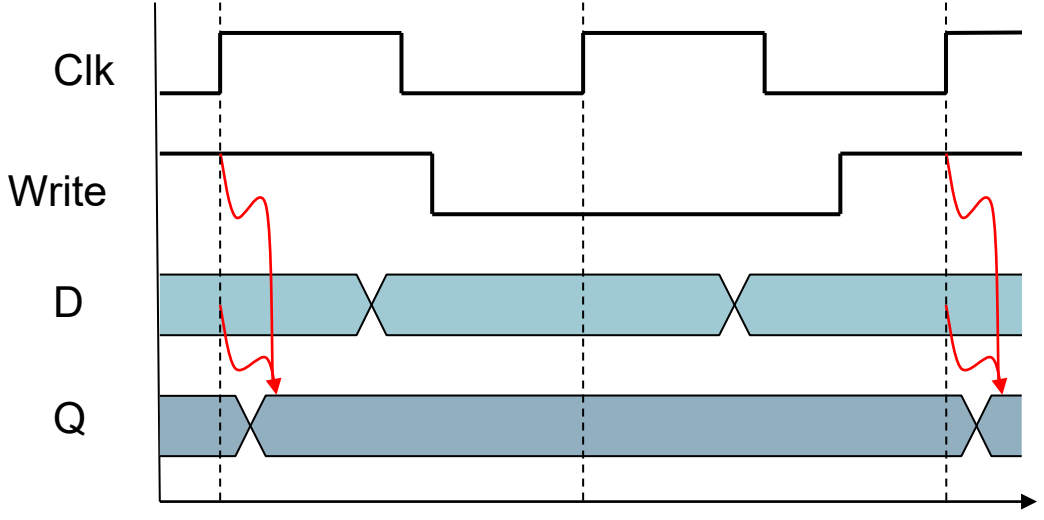


Sequential Elements

Register with write control

- Only updates on clock edge when write control input is 1
- Used when stored value is required later

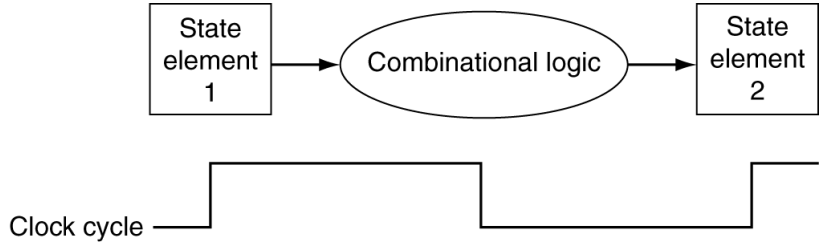


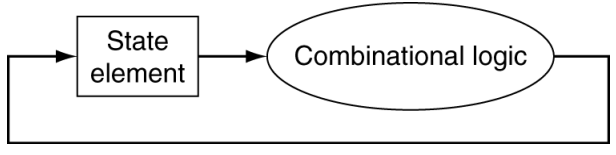


Clocking Methodology

Combinational logic transforms data during clock cycles

- Between clock edges
- Input from state elements, output to state element
- Longest delay determines clock period





Building a Datapath

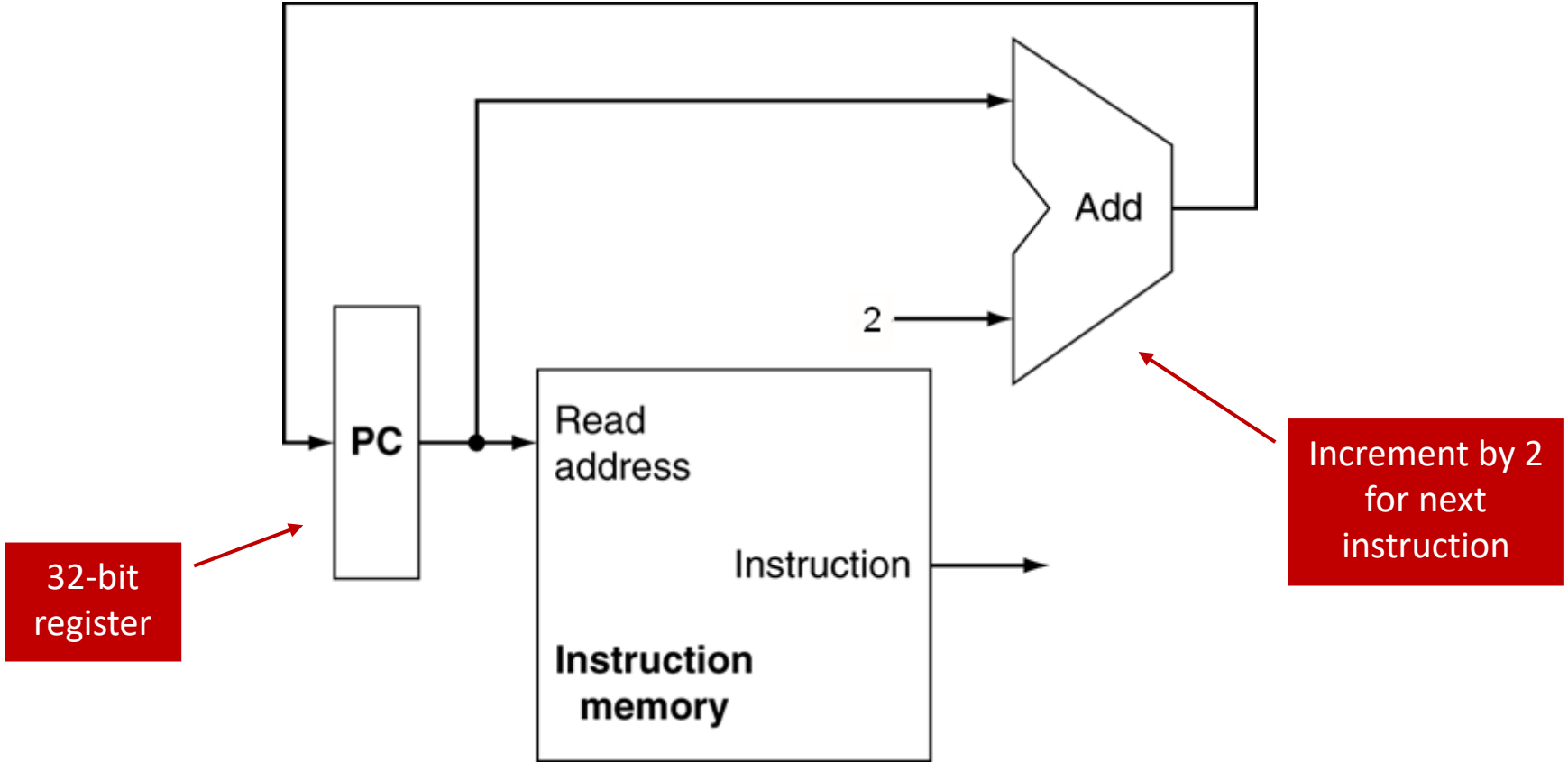
Datapath

- Elements that process data and addresses in the CPU
  - Registers, ALUs, mux's, memories, …

We will build a LEGv7-M datapath incrementally

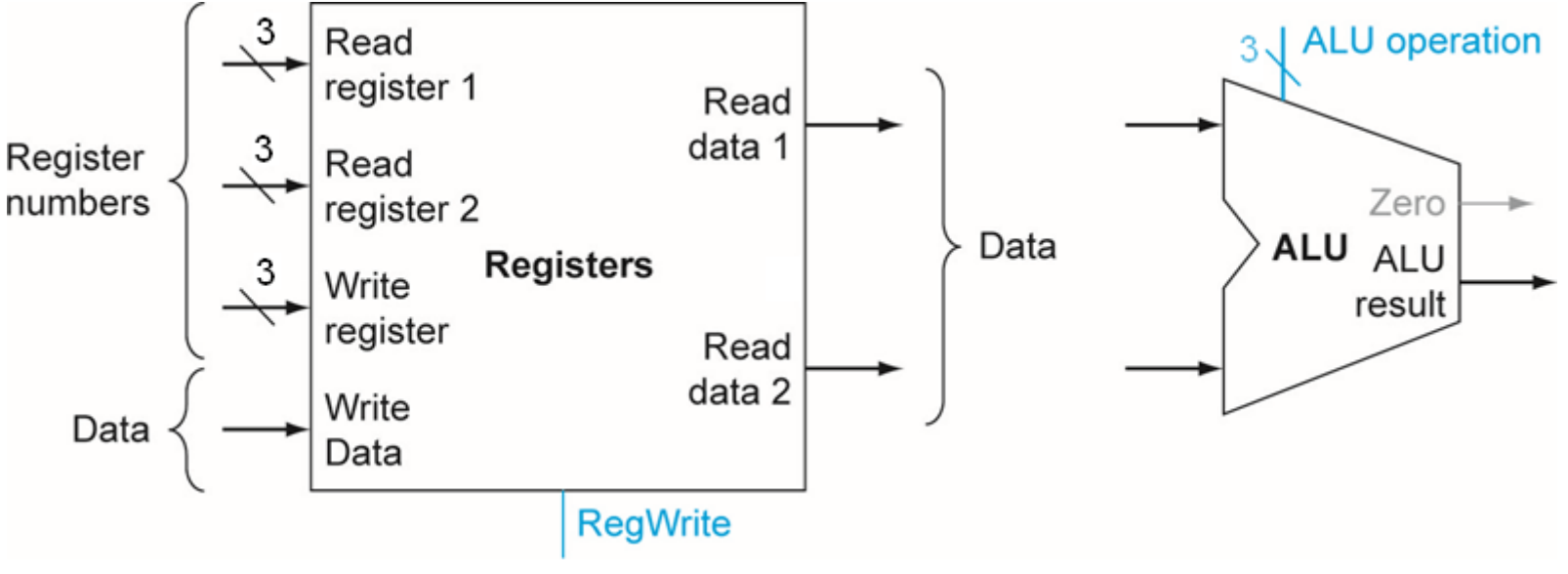
- Refining the overview design

Instruction Fetch



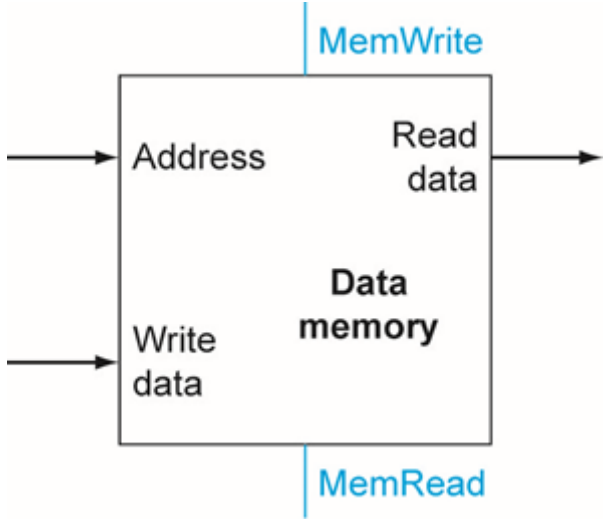
R-type Instructions ADDS, SUBS, ANDS, ORRS

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



M-type Instructions LDR, STR

- Read register operands
- Calculate address using 5-bit offset
  - Shift left 2 places (word displacement)
  - Use ALU
- LDR: Read memory and update register
- STR: Write register value to memory

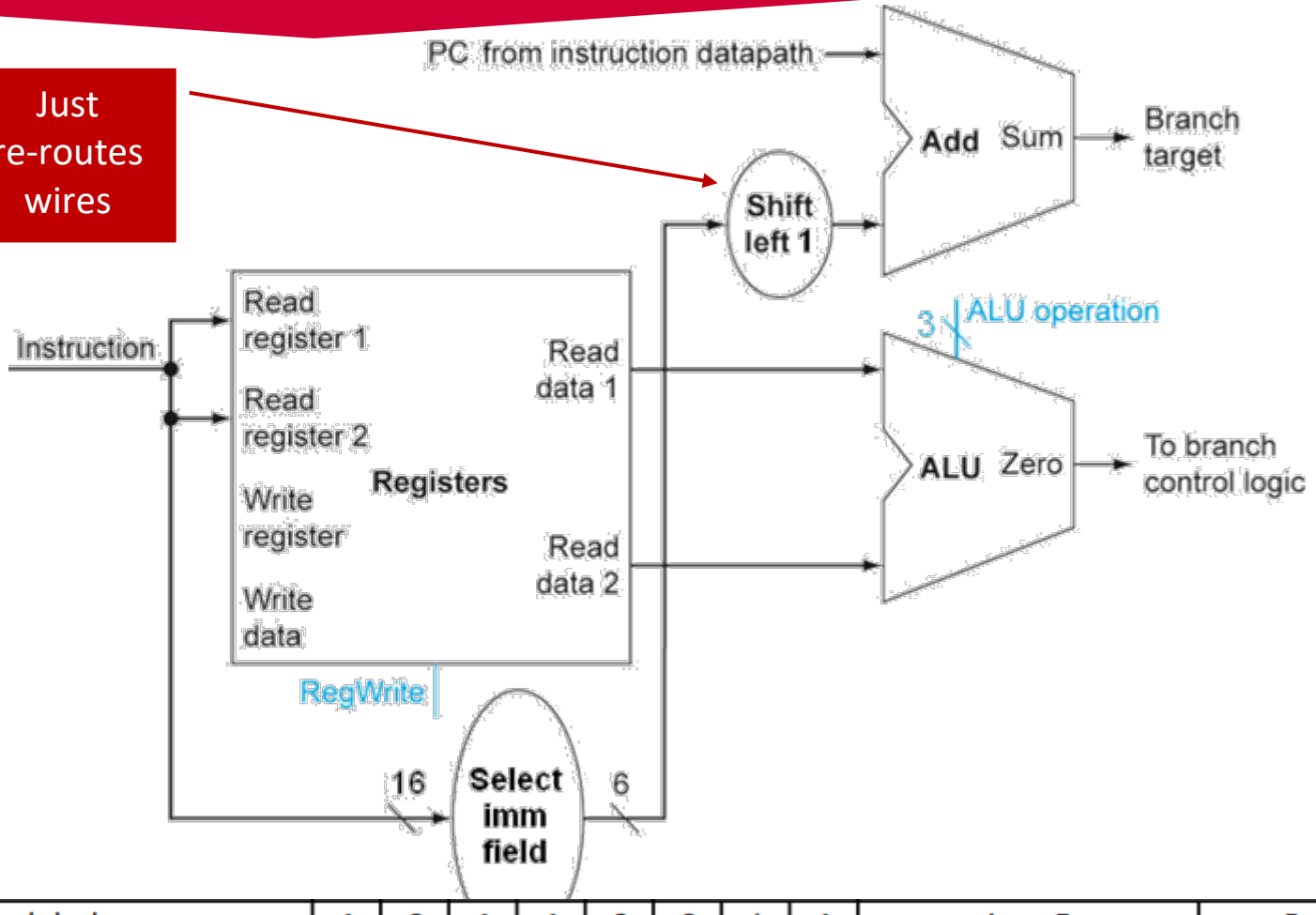


CB-type Instruction CBZ

- Read register operand
- Compare operand to zero
  - Use ALU pass input and check Zero output
- Calculate target address
  - 6-bit displacement field
  - Shift left 1 place (half word displacement)
  - Add to PC

CB-type Instruction CBZ

Just re-routes wires



CBZ.N <Rn>,<label>	1	0	1	1	0	0	i	1	imm5	Rn

Composing the Elements

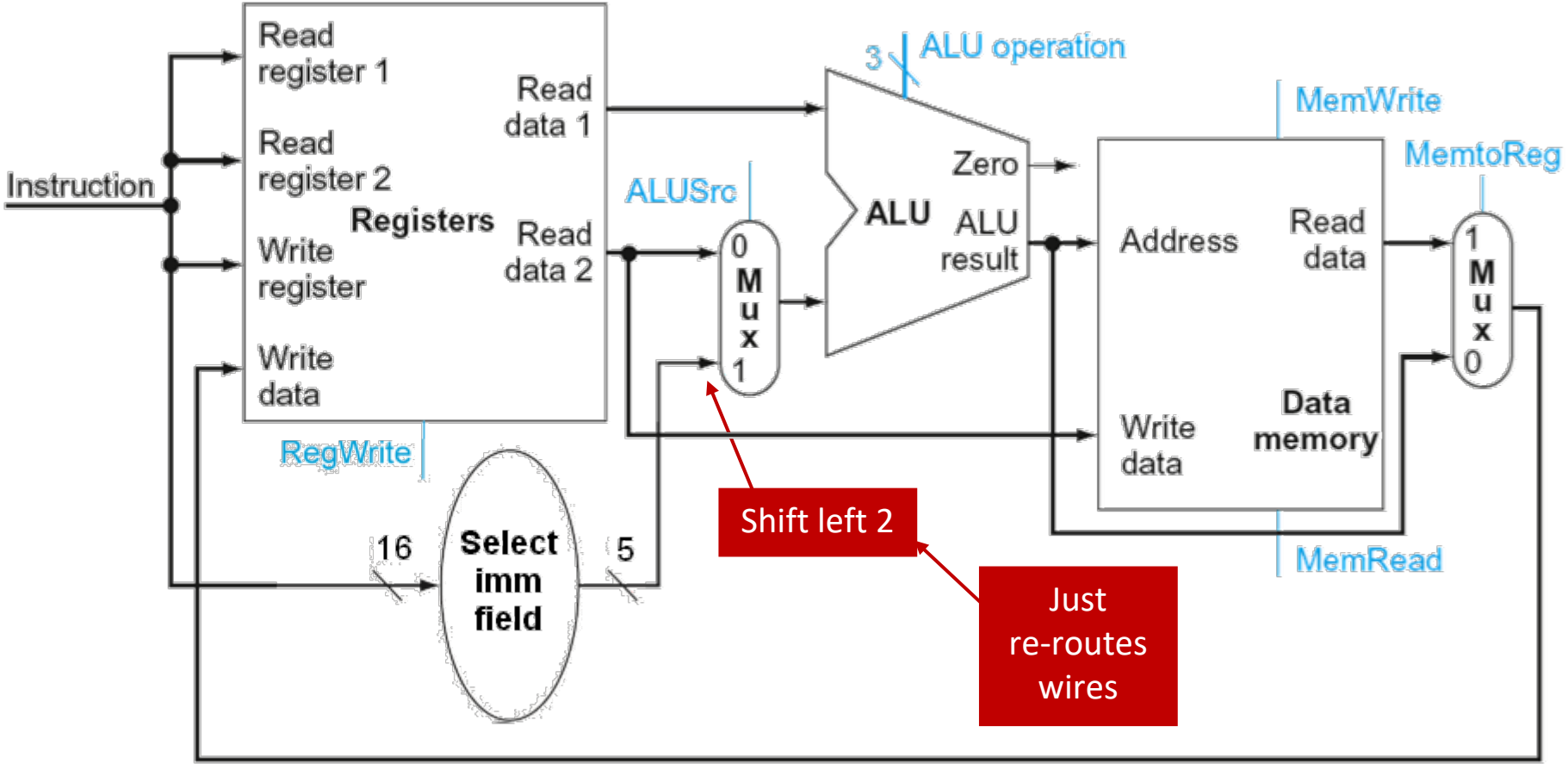
First-cut data path does an instruction in one clock cycle

- Each datapath element can only do one function at a time
- Hence, we need separate instruction and data memories

Use multiplexers where alternate data sources are used for different instructions

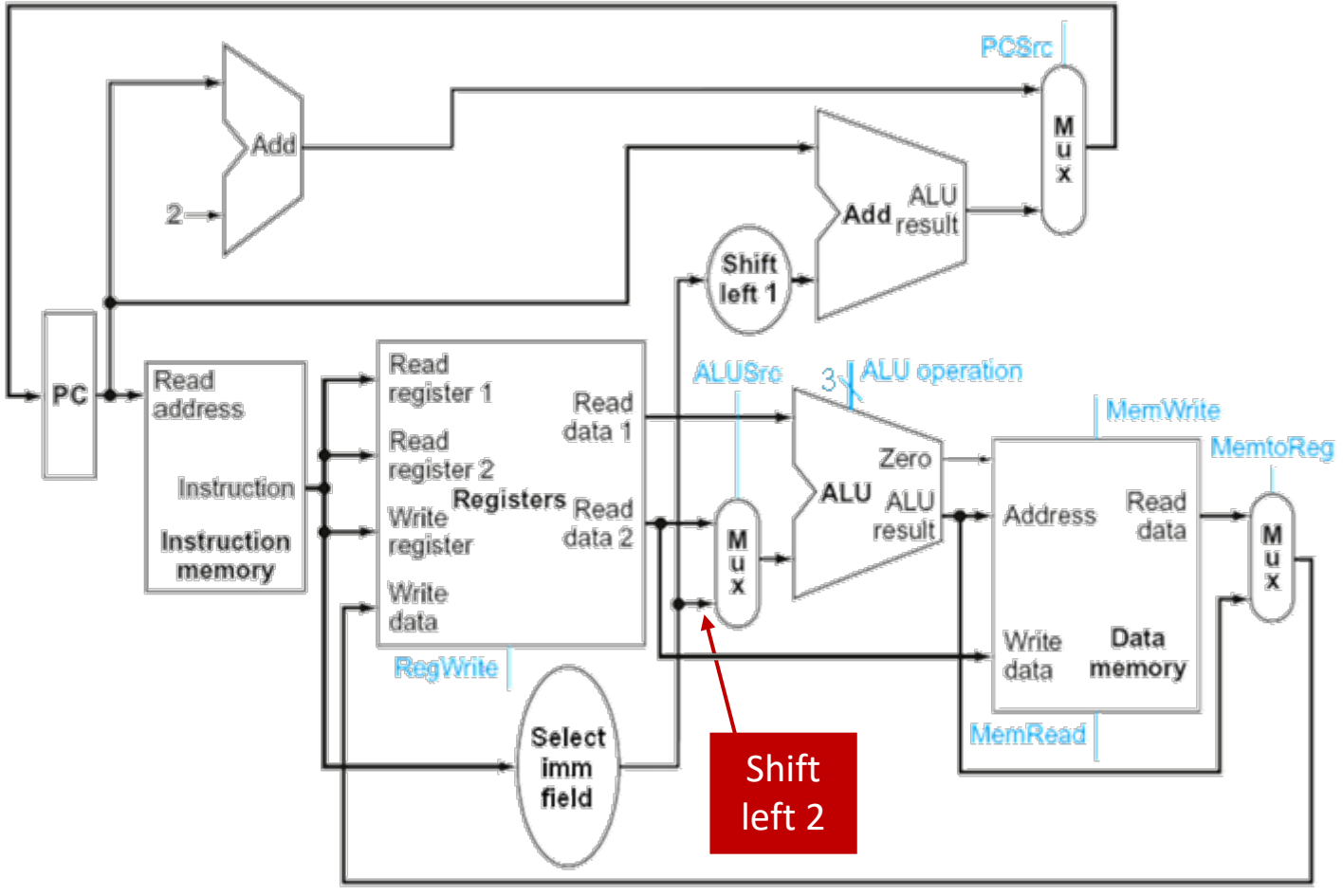
M-Type/D-Type Datapath

ADDS, SUBS, ANDS, EORS, ORRS, LDR, STR



Full Datapath

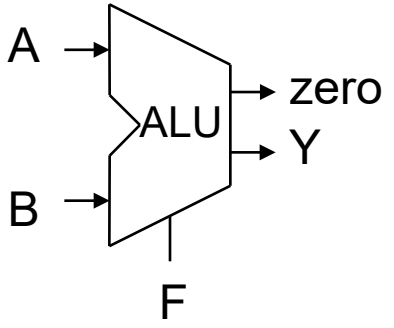
ADDS, SUBS, ANDS, EORR, ORRS, LDR, STR, CBZ



ALU Control

ALU used for

- R-type ADDS, SUBS, ANDS, EORS, ORRS:
  - Function depends on opcode
- M-type LDR, STR:
  - Function = add
- CB-type CBZ:
  - Function = pass



Function	F
ADD	10X
SUB	11X
AND	000
EOR	001
OR	010
Pass	011

Coding of F is not random, will be explained later

ALU Control

How to determine instruction type from opcodes?

Instructie	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	0	Rm			Rn			Rd		
SUBS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	1	Rm			Rn			Rd		
ANDS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	0	Rm			Rdn		
EORS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	1	Rm			Rdn		
ORRS.N <Rdn>,<Rm>	0	1	0	0	0	0	1	1	0	0	Rm			Rdn		
STR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	0	imm5					Rn			Rt		
LDR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	1	imm5					Rn			Rt		
CBZ.N <Rn>,<label>	1	0	1	1	0	0	i	1	imm5					Rn		

ALU Control

Determine instruction type from opcodes

Instructie	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	0	Rm			Rn			Rd		
SUBS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	1	Rm			Rn			Rd		
ANDS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	0	Rm			Rdn		
EORS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	1	Rm			Rdn		
ORRS.N <Rdn>,<Rm>	0	1	0	0	0	0	1	1	0	0	Rm			Rdn		
STR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	0	imm5					Rn			Rt		
LDR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	1	imm5					Rn			Rt		
CBZ.N <Rn>,<label>	1	0	1	1	0	0	i	1	imm5					Rn		

Type	ALUOp
R-type	00
M-type	01
CB-type	11

ALU Control

How to determine F for R-type from opcodes?

Instructie	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	0	Rm			Rn			Rd		
SUBS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	1	Rm			Rn			Rd		
ANDS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	0	Rm			Rdn		
EORS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	1	Rm			Rdn		
ORRS.N <Rdn>,<Rm>	0	1	0	0	0	0	1	1	0	0	Rm			Rdn		

Function	F
ADD	10X
SUB	11X
AND	000
EOR	001
OR	010
Pass	011

ALU Control

Determine F for R-type from opcodes

Instructie	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	0	Rm			Rn			Rd		
SUBS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	1	Rm			Rn			Rd		
ANDS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	0	Rm			Rdn		
EORS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	1	Rm			Rdn		
ORRS.N <Rdn>,<Rm>	0	1	0	0	0	0	1	1	0	0	Rm			Rdn		

Function	F
ADD	10X
SUB	11X
AND	000
EOR	001
OR	010
Pass	011

Only remaining code

ALU Control

2-bit ALUOp derived from opcode

- Combinational logic derives ALU control (F)

Opcode	Type	ALUOp	ALU-function	ALU-control
ADDS	R-type	00	ADD	10X
SUBS			SUB	11X
ANDS			AND	000
EORS			EOR	001
ORRS			OR	010
STR	M-type	01	ADD	10X
LDR			ADD	10X
CBZ	CB-type	11	Pass	011

The Main Control Unit

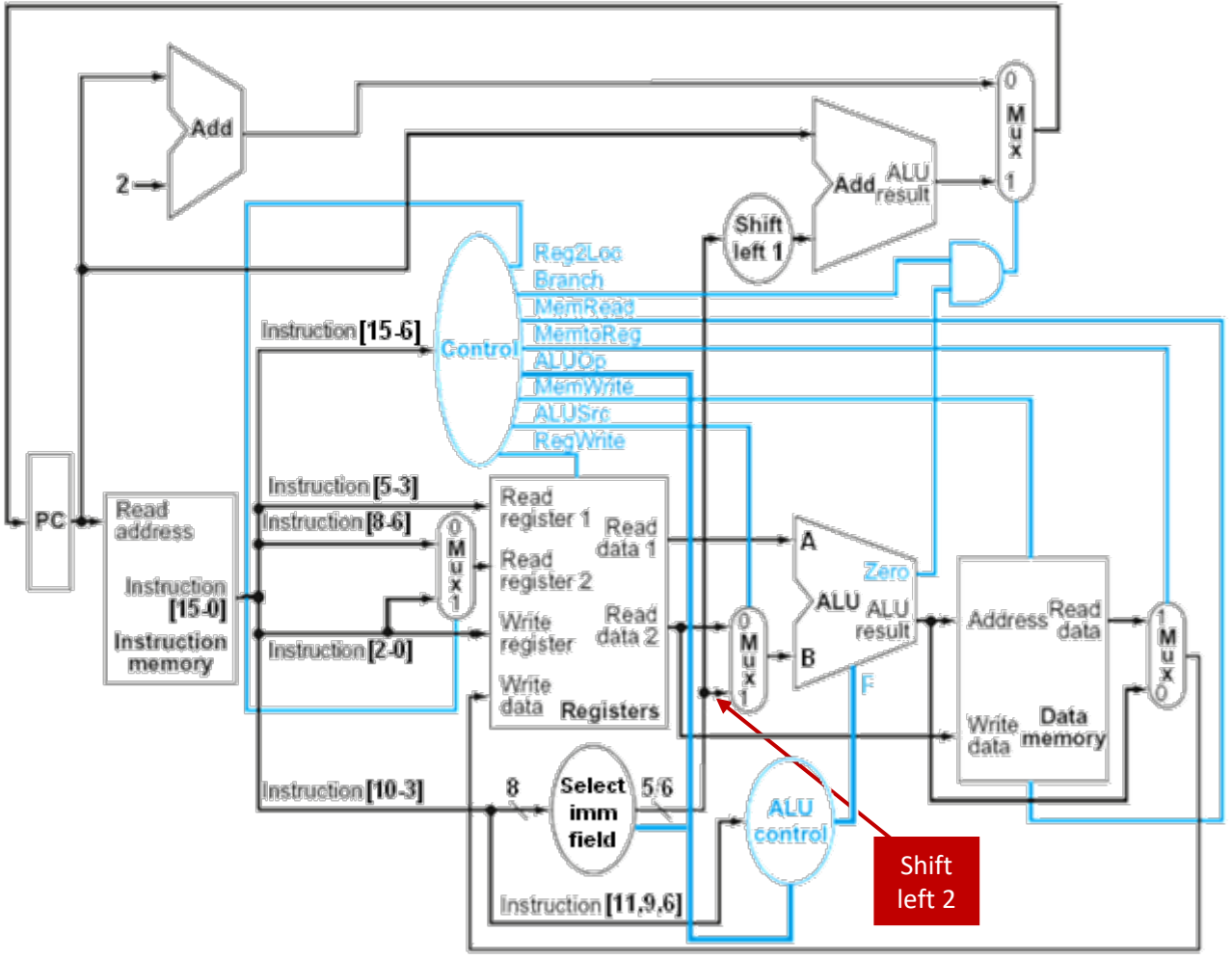
All control signals are derived from machine codes

Instructie	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	0	Rm			Rn			Rd		
SUBS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	1	Rm			Rn			Rd		
ANDS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	0	~~Rm~~ n			~~Rdn~~ m		
EORS.N <Rdn>,<Rm>	0	1	0	0	0	0	0	0	0	1	~~Rm~~ n			~~Rdn~~ m		
ORRS.N <Rdn>,<Rm>	0	1	0	0	0	0	1	1	0	0	~~Rm~~ n			~~Rdn~~ m		
STR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	0	imm5					Rn			Rt		
LDR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	1	imm5					Rn			Rt		
CBZ.N <Rn>,<label>	1	0	1	1	0	0	i	1	imm5					~~Rn~~ m		

Make decoding simpler:

- Exchange Rm and Rn for logic instructions
- Use Rm for CBZ (ALU should pass input B)

Datapath With Control



Control signals

Signal name	Effect when deasserted	Effect when asserted
Reg2Loc	The register number for Read register 2 comes from the Rm field (bits 8-6).	The register number for Read register 2 comes from the Rt / Rm field (bits 2-0).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is imm5 field of the instruction shift left 2.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 2.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Exercise

What are the values of control signals generated by the control for the instructions:

- `SUBS R0, R1, R2`
- `ANDS R0, R1`
- `LDR  R0, [R1, #4]`
- `STR  R0, [R1, #8]`
- `CBZ  R0, label`

Exercise

When silicon chips are fabricated, defects in materials (e.g., silicon) and manufacturing errors can result in defective circuits. A very common defect is for one signal wire to get “broken” and always register a logical 0. This is often called a “stuck-at-0” fault.

Which instructions fail to operate correctly:

- if the MemToReg wire is stuck at 0?
- if the ALUSrc wire is stuck at 0?
- if the Reg2Loc wire is stuck at 0?

Implementing B-type (B)

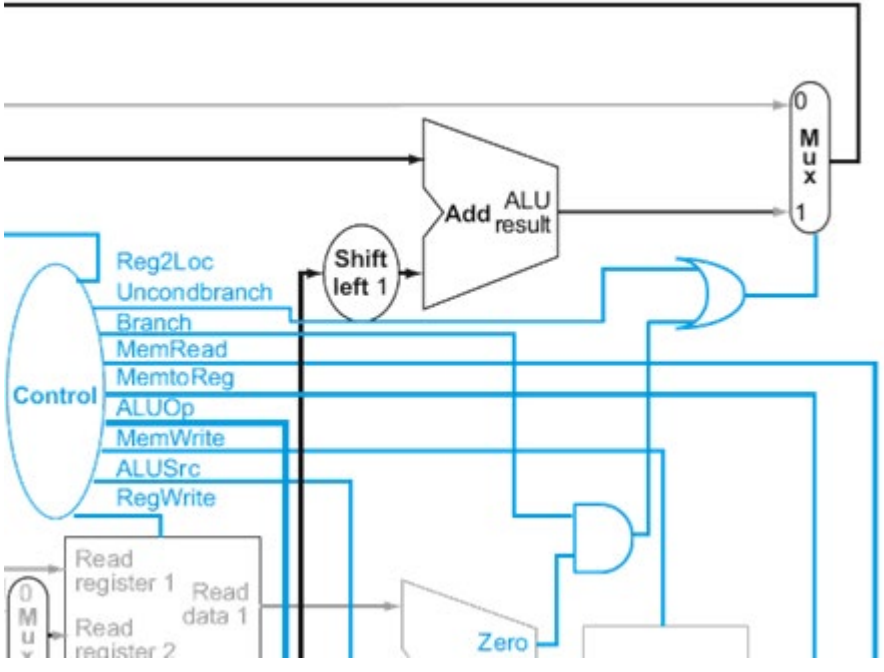
Instructie	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B.N <label>	1	1	0	1	1	1	1	0	imm8_two's_complement							

- Branch uses half word (16-bit) offset
- PC ← PC + (sign extended offset) x 2
- Need an extra control signal decoded from opcode

Datapath With B Added

Changes:

- The imm8 two's complement offset must be sign extended to 31 bits when B instruction is detected



Exercise

Problems in this exercise assume that the logic blocks used to implement a processor's datapath have the following latencies:

Select imm field

I-Mem / D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

"Register read" is the time needed after the rising clock edge for the new register value to appear on the output. This value applies to the PC only. "Register setup" is the amount of time a register's data input must be stable before the rising edge of the clock. This value applies to both the PC and Register File.

- Although the control unit as a whole requires 50 ps, it so happens that we can extract the correct value of the Reg2Loc control wire directly from the instruction. Thus, the value of this control wire is available at the same time as the instruction. Explain how we can extract this value directly from the instruction. Hints: Exchange the inputs of the mux. Carefully examine the opcodes.

Exercise

Problems in this exercise assume that the logic blocks used to implement a processor's datapath have the following latencies:

Select imm field

I-Mem / D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

- What is the latency of an R-type instruction (i.e., how long must the clock period be to ensure that this instruction works correctly)?
- What is the latency of LDR? (Check your answer carefully. Many students place extra muxes on the critical path.)
- What is the latency of STR? (Check your answer carefully. Many students place extra muxes on the critical path.)
- What is the latency of CBZ?
- What is the latency of B?
- What is the minimum clock period for this CPU?

Performance Issues

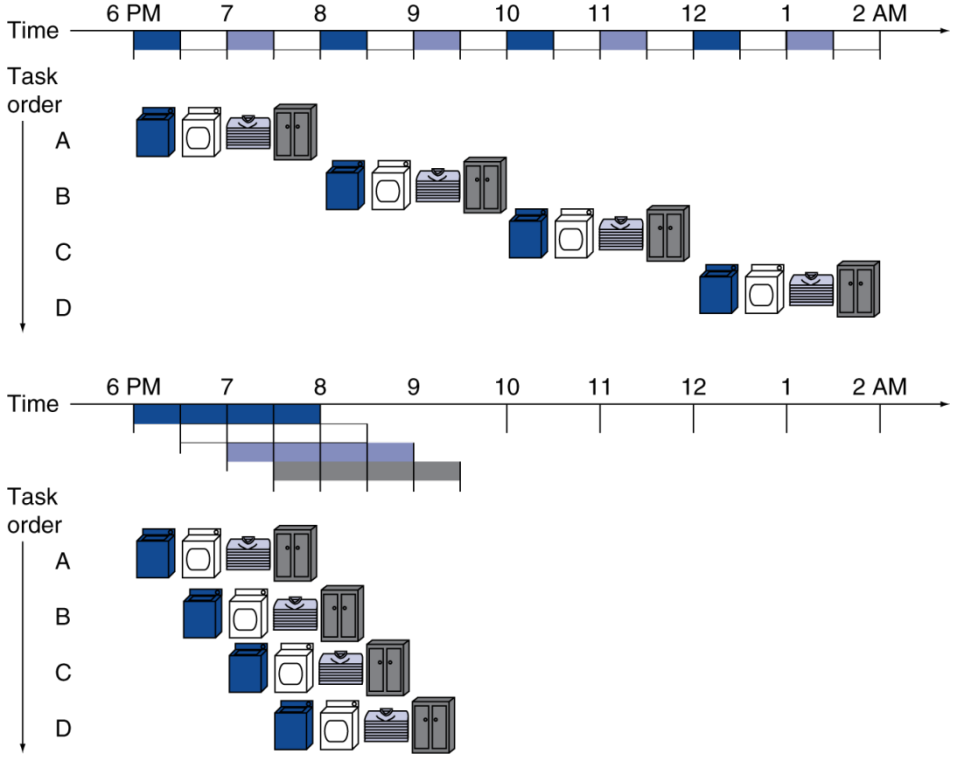
- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

Agenda

- A one cycle implementation of LEGv7-M
- A pipelined implementation of LEGv7-M
- Branch prediction
- Cache memory

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



Speedup?

- Four loads:
  - Speedup
    = 8/3.5 = 2.3
- Non-stop:
  - Speedup
    = 2n/0.5n + 1.5 ≈ 4
    = number of stages

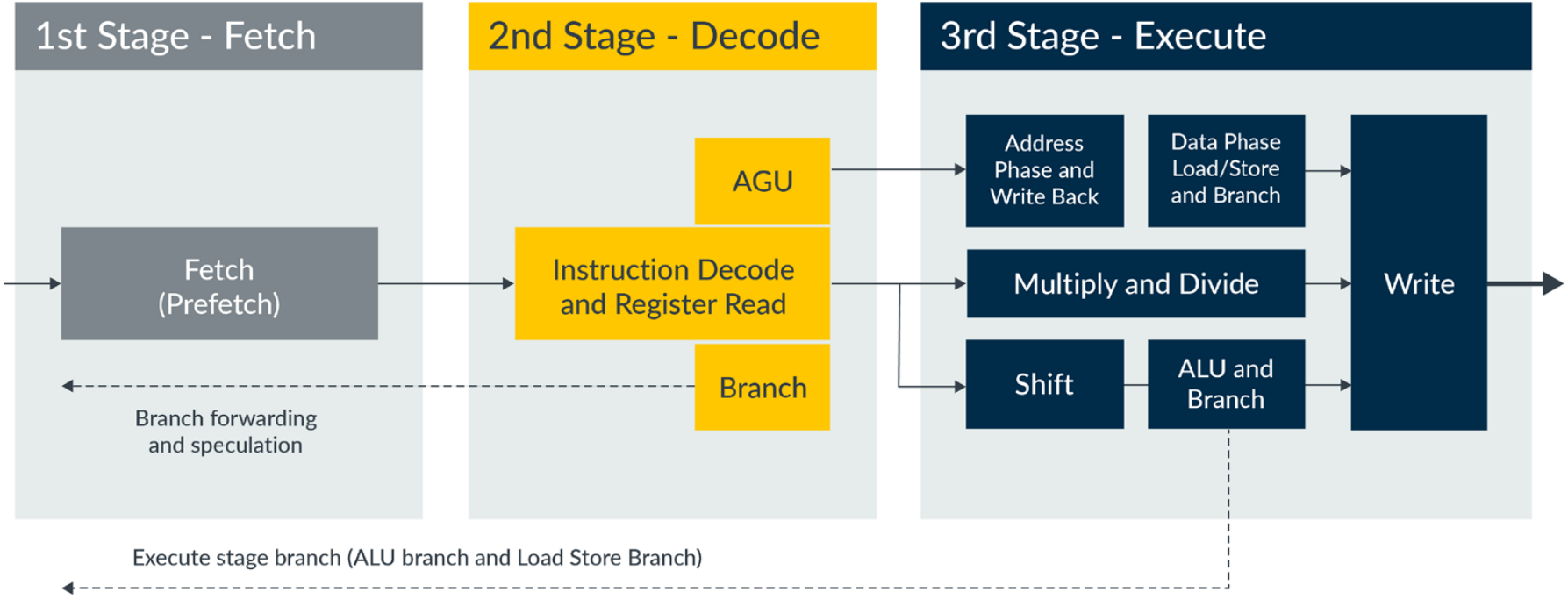
LEGv7M Pipeline

Five stages, one step per stage

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

ARMv7 Cortex-M4 Pipeline has only three stages

Cortex-M4 Pipeline



Bron: [Arm Cortex-M4 Datasheet]

Pipeline Performance

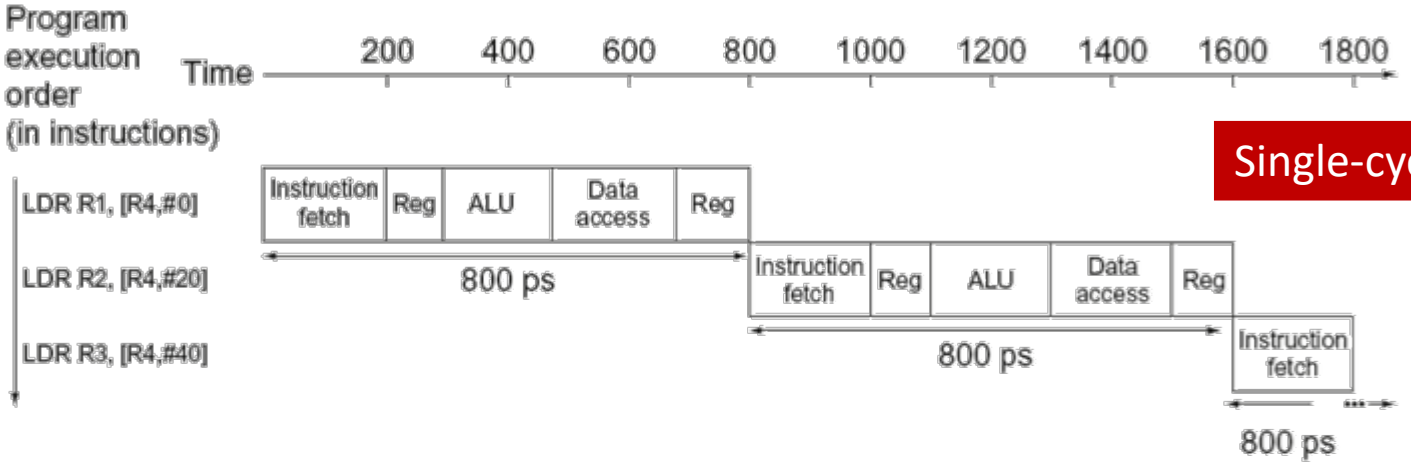
Assume time for stages is

- 100 ps for register read or write
- 200 ps for other stages

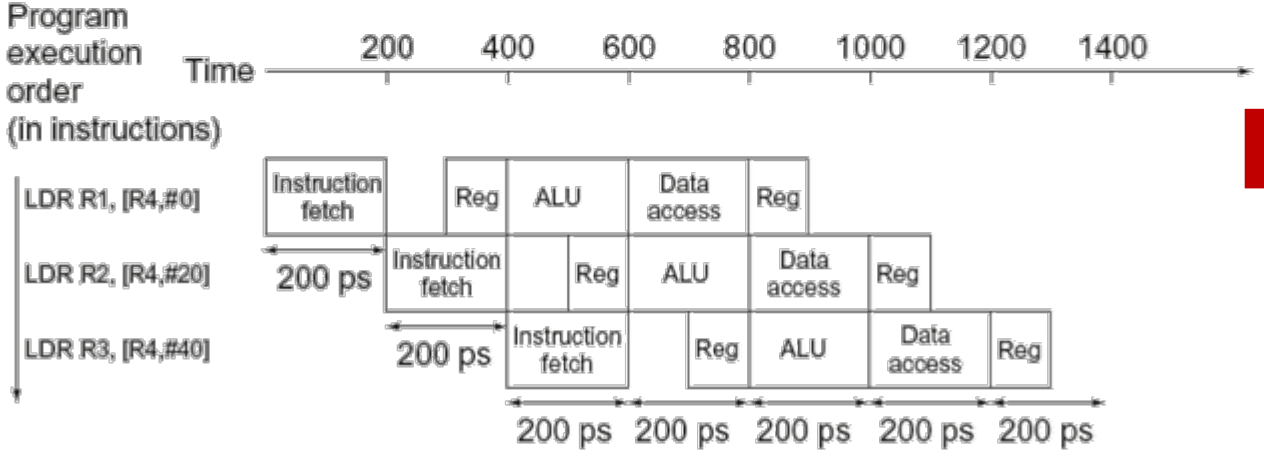
Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
LDR	200ps	100 ps	200ps	200ps	100 ps	800ps
STR	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
CBZ	200ps	100 ps	200ps			500ps

Pipeline Performance



Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)

Pipeline Speedup

If all stages are balanced

- i.e., all take the same time
- Time between instructions$_{pipelined}$

$$= \frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$$

If not balanced, speedup is less

Speedup due to increased throughput

- Latency (time for each instruction) does not decrease

Pipelining and ISA Design

LEGv7-M ISA designed for pipelining

- All instructions are 16-bits
  - Easier to fetch and decode in one cycle
  - c.f. x86: 1- to 17-byte instructions
- Few and regular instruction formats
  - Can decode and read registers in one step
- Load/store addressing
  - Can calculate address in 3rd stage, access memory in 4th stage
- Alignment of memory operands
  - 32-bit Memory access takes only one cycle

Hazards

Situations that prevent starting the next instruction in the next cycle

- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structure Hazards

Conflict for use of a resource

- In LEGv7-M pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories (Harvard architecture)
  - Or separate instruction/data caches

Structure Hazards

Conflict for use of a resource

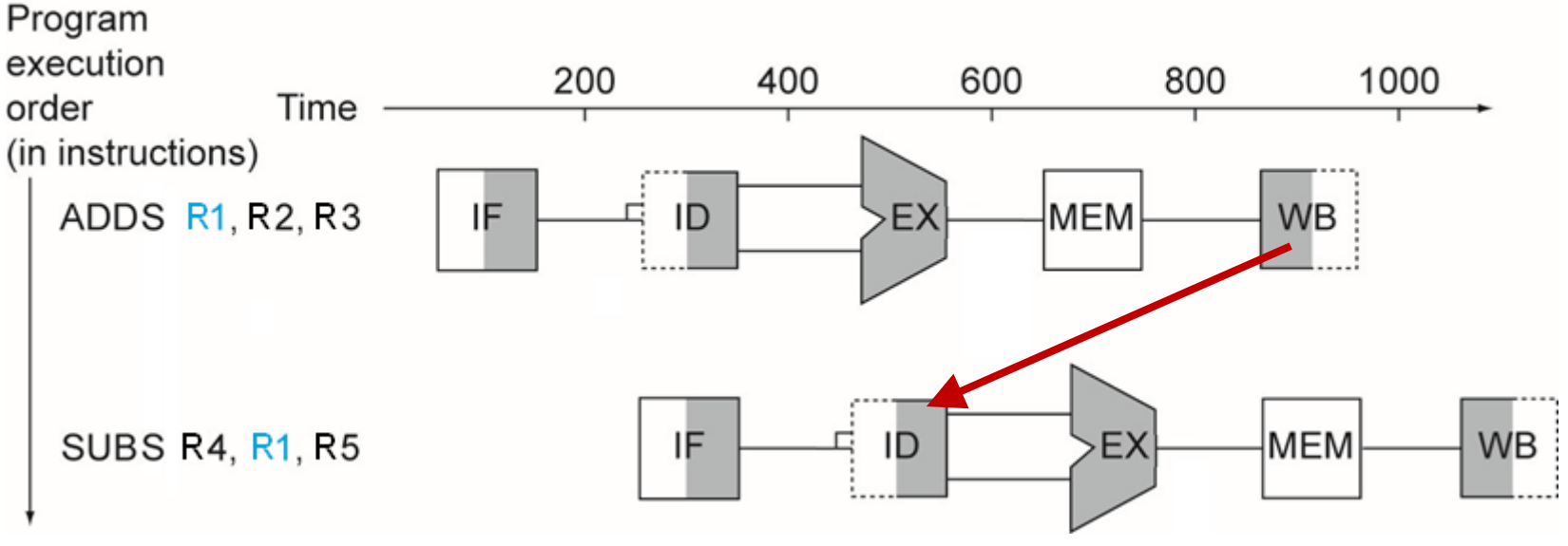
- In LEGv7-M the registers are used in stages ID (read) and WB (write)
- Assumed time for stages is
  - 100 ps for register read or write
  - 200 ps for other stages
- Hence, write and read can be combined in one pipeline slot
  - write in first 100 ps, read in last 100 ps

Data Hazards

An instruction depends on completion of data access by a previous instruction

- ADDS R1, R2, R3
  SUBS R4, R1, R5

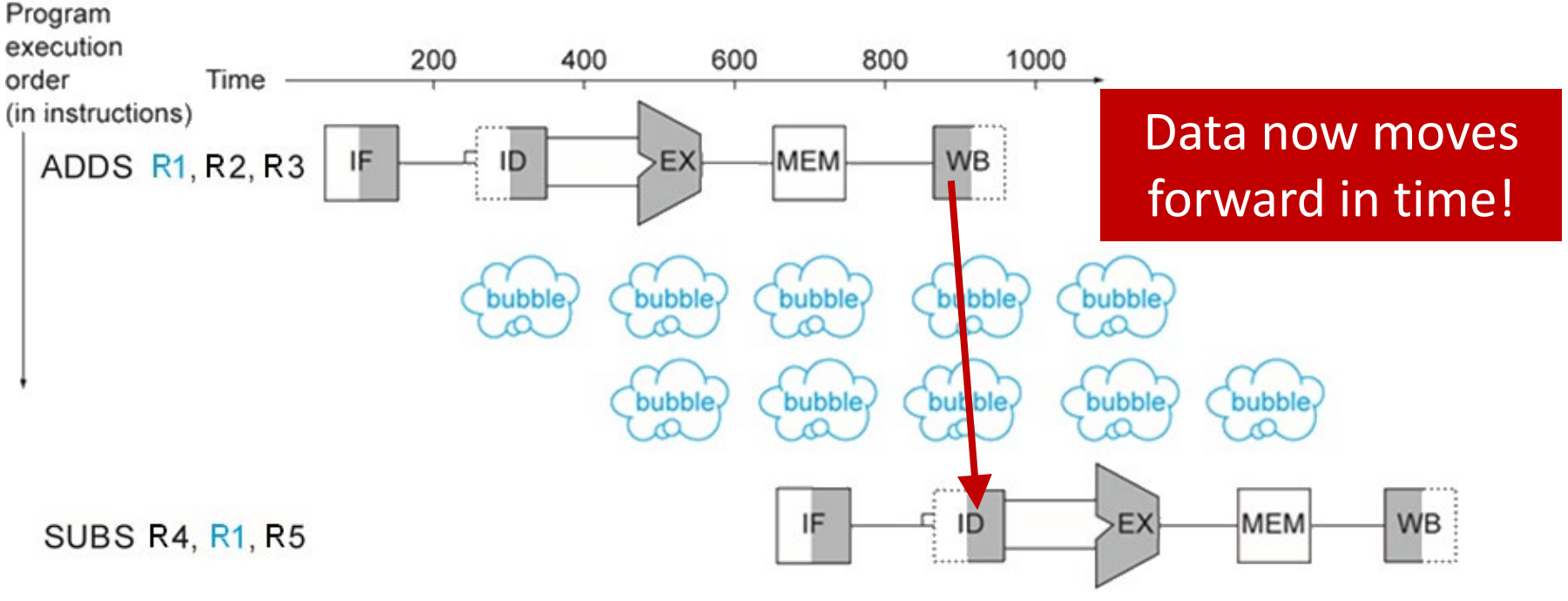
Data can not move backward in time!



Stalling (aka insert bubbles)

Wait for the data to be stored in a register

- Requires extra control logic



Exercise

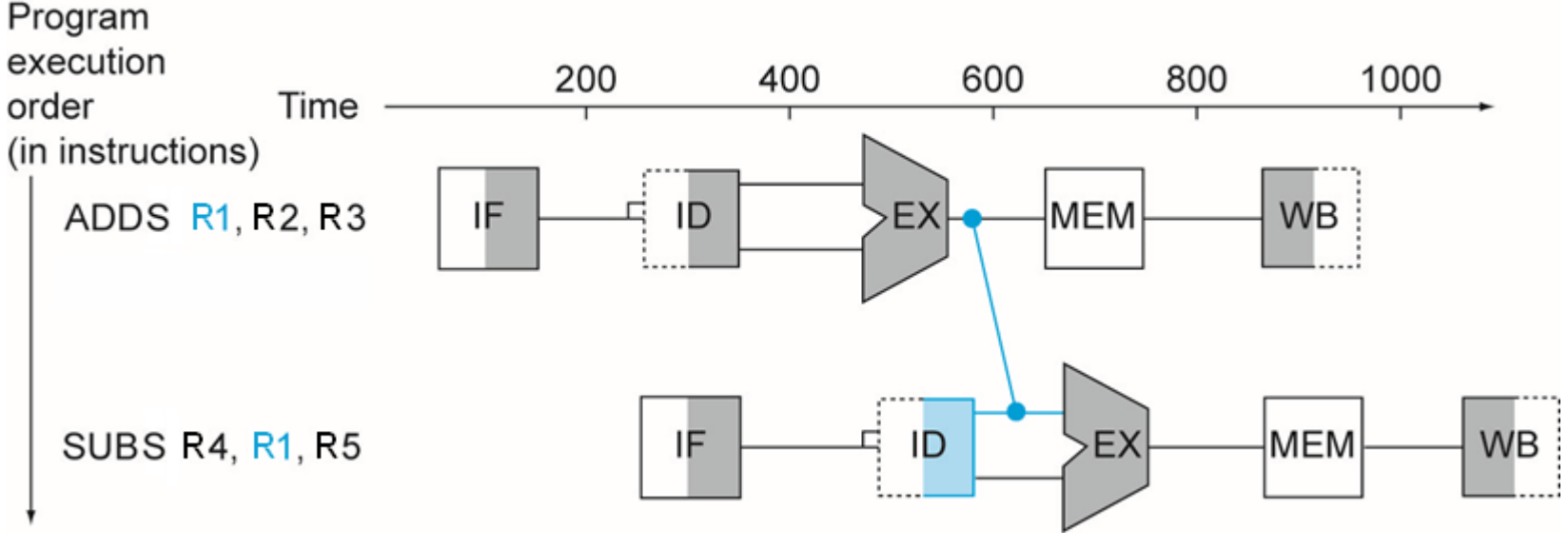
Were must the processor insert bubbes (NOPs) when the following code is run on the pipeline:

```
ADDS R1, R2, R3
ADDS R4, R1, R2
ADDS R5, R1, R6
ADDS R7, R1, R4
```

Forwarding (aka Bypassing)

Use result when it is computed

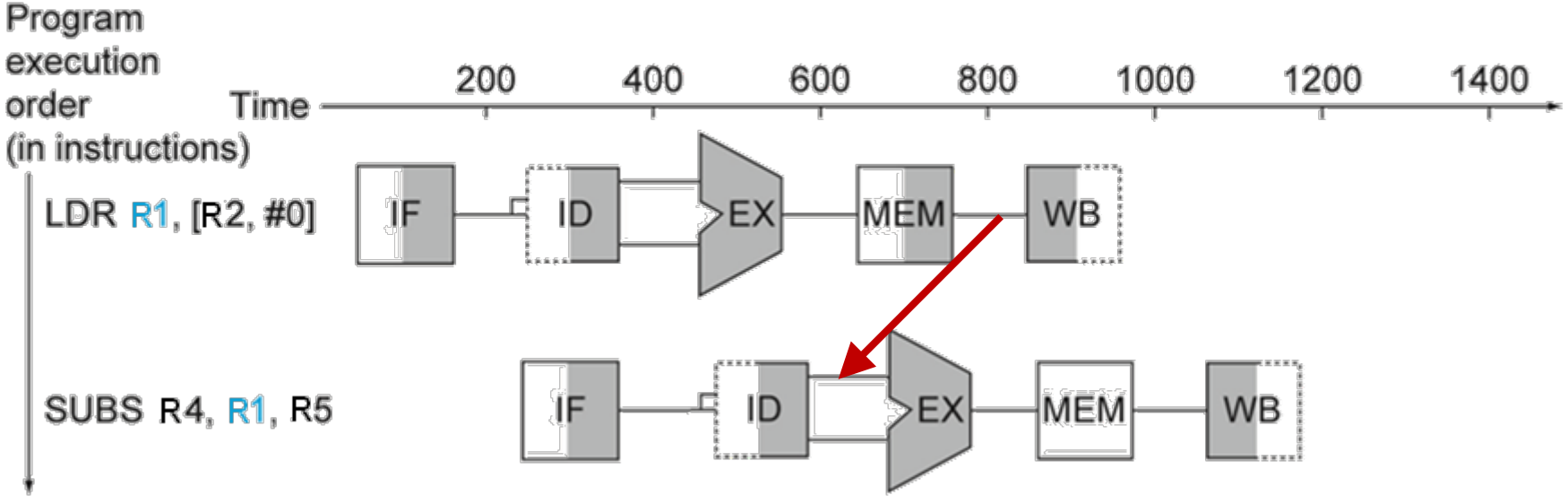
- Don’t wait for it to be stored in a register
- Requires extra connections in the datapath and extra control logic



Load-Use Data Hazard

Can’t always avoid stalls by forwarding

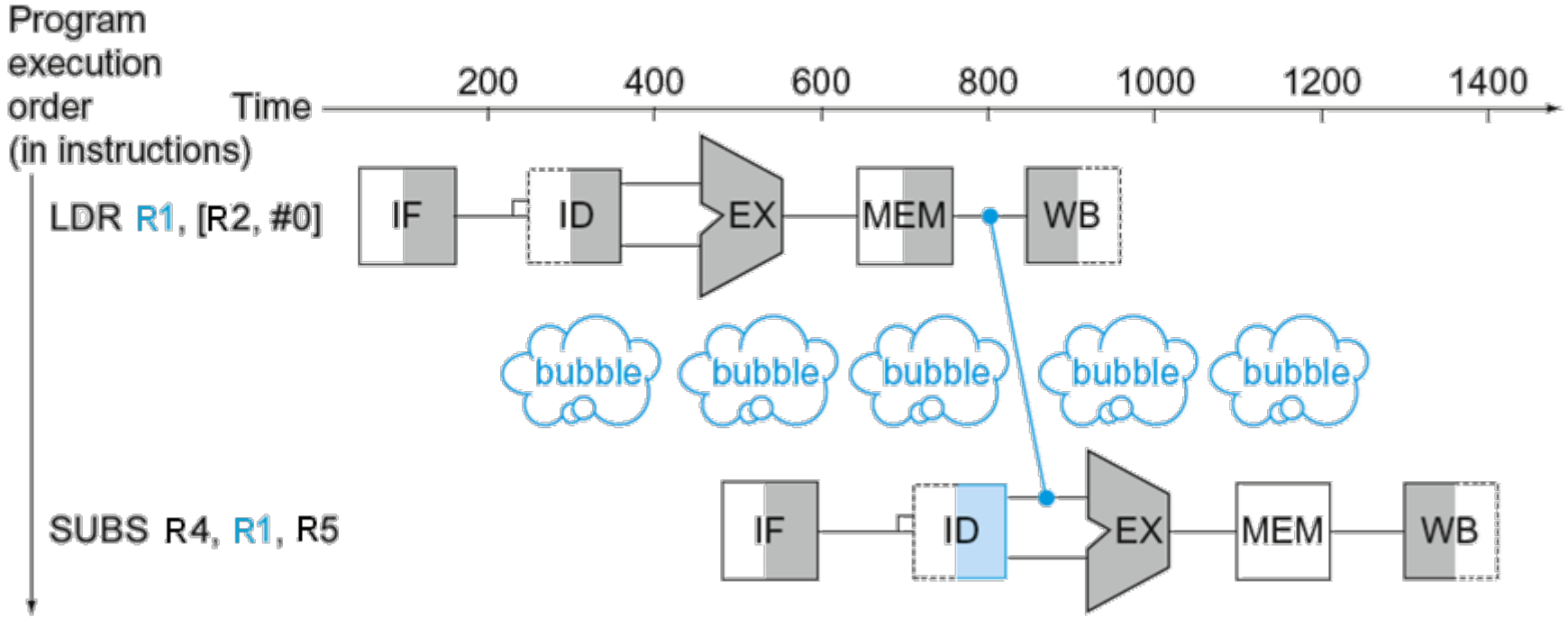
- If value not computed when needed
- Can’t forward backward in time!



Load-Use Data Hazard

Can’t avoid stalls by forwarding

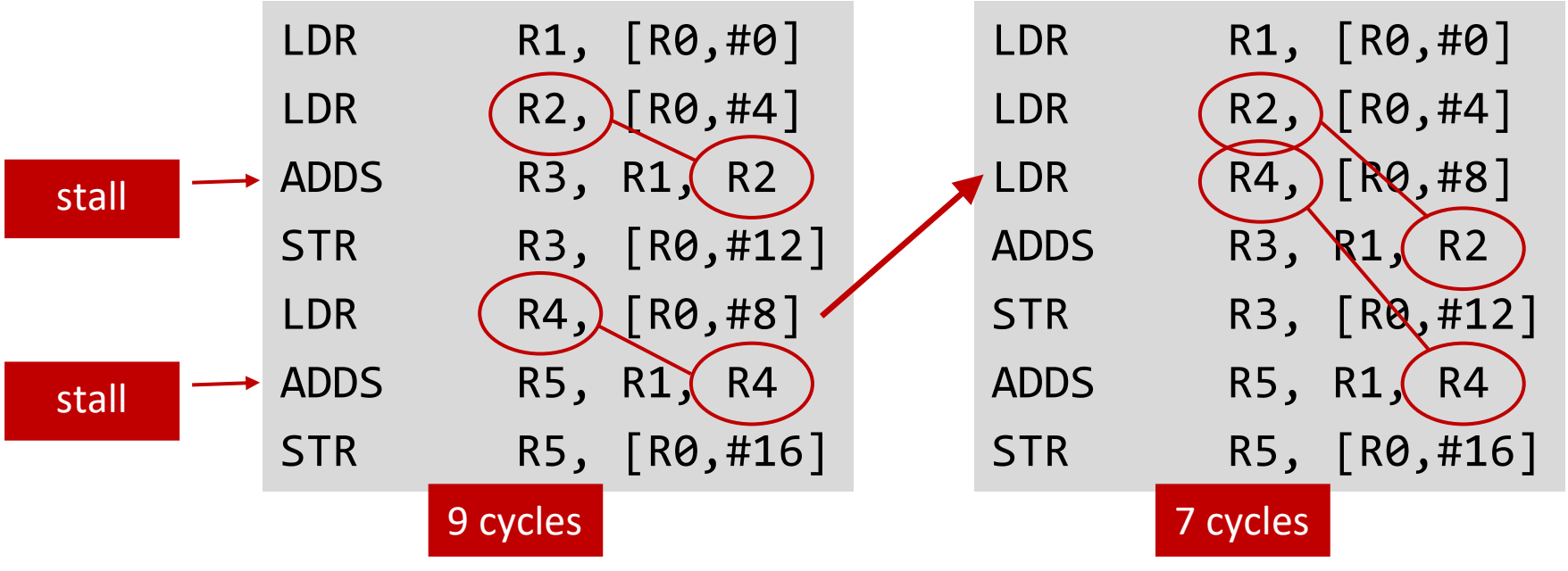
- Stall can be limited to one cycle by using forwarding.



Code Scheduling to Avoid Stalls

Reorder code to avoid use of load result in the next instruction

C code for `A[3]=A[0]+A[1]; A[4]=A[0]+A[2]; // A in R0`



Control Hazards

Branch determines flow of control

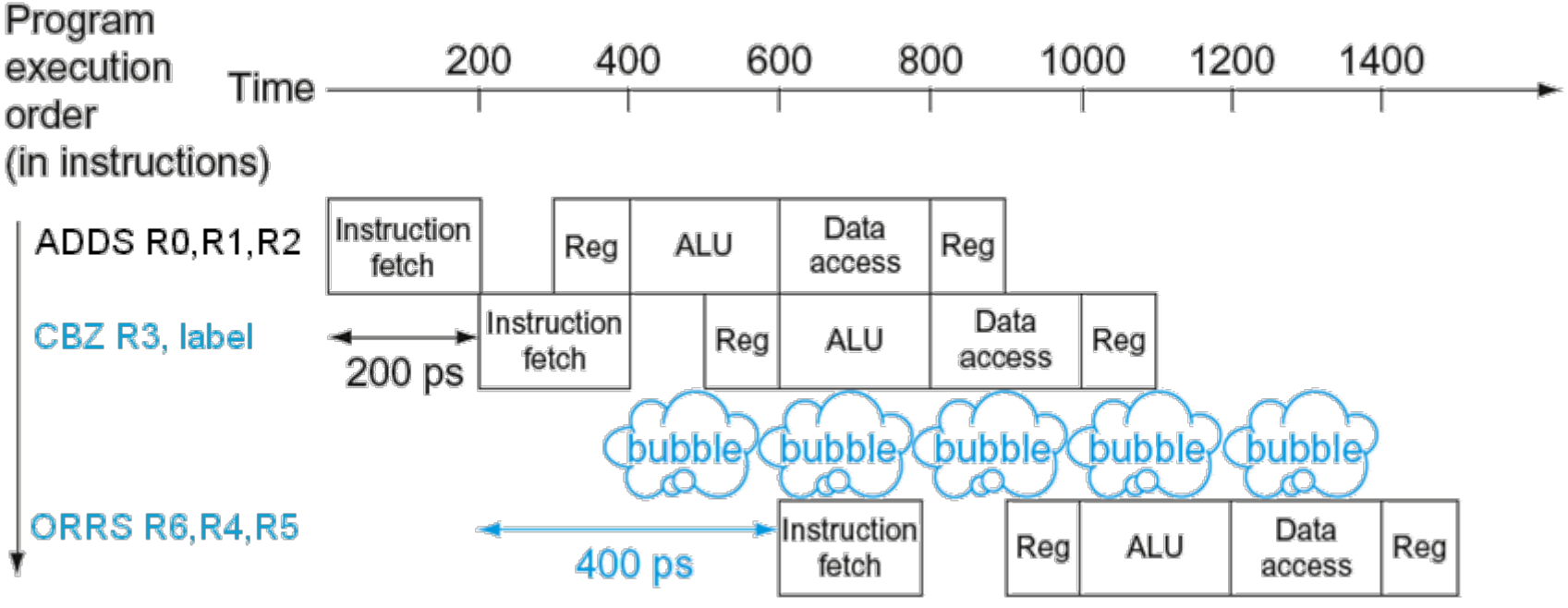
- Fetching next instruction depends on branch outcome
- Pipeline can't always fetch correct instruction
  - Still working on ID stage of branch

In LEGv7-M pipeline

- Need to compare registers and compute target early in the pipeline
- Add hardware to do it in ID stage

Stall on Branch

Wait until branch outcome determined before fetching next instruction



Agenda

- A one cycle implementation of LEGv7-M
- A pipelined implementation of LEGv7-M
- Branch prediction
- Cache memory

Branch Prediction

Longer pipelines can't readily determine branch outcome early

- Stall penalty becomes unacceptable

Predict outcome of branch

- Only stall if prediction is wrong

In LEGv7-M pipeline

- Can predict branches not taken
- Fetch instruction after branch, with no delay

More-Realistic Branch Prediction

Static branch prediction

- Based on typical branch behavior
- Example: loop and if-statement branches
  - Predict backward branches taken
  - Predict forward branches not taken

Dynamic branch prediction

- Hardware measures actual branch behavior
  - e.g., record recent history of each branch
- Assume future behavior will continue the trend
  - When wrong, stall while re-fetching, and update history

Dynamic Branch Prediction

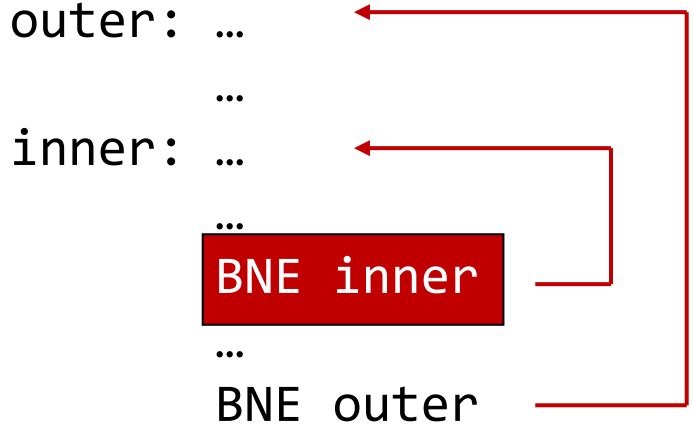
In deeper and superscalar pipelines, branch penalty is more significant

Use dynamic prediction

- Branch prediction buffer (aka branch history table)
- Indexed by recent branch instruction addresses
- Stores outcome (taken/not taken)
- To execute a branch
  - Check table, expect the same outcome
  - Start fetching from fall-through or target
  - If wrong, flush pipeline and flip prediction

1-Bit Predictor: Shortcoming

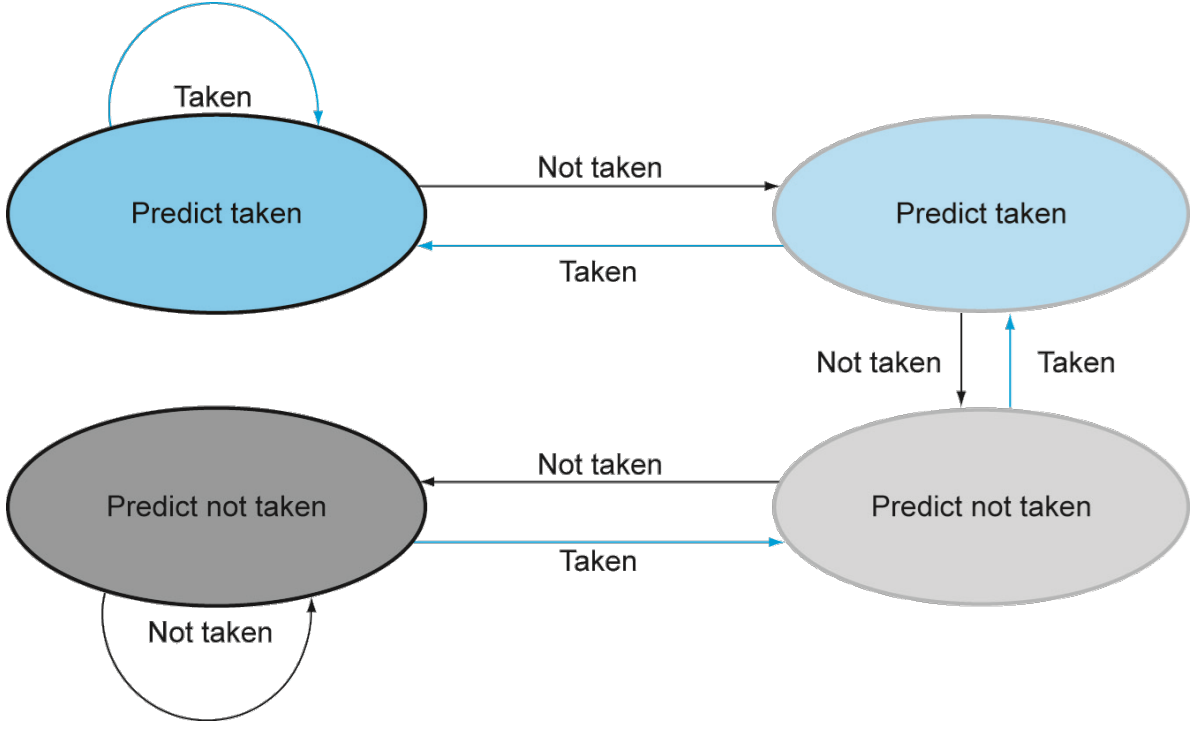
Inner loop branches mispredicted twice!



- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

Only change prediction on two successive mispredictions



Calculating the Branch Target

Even with predictor, still need to calculate the target address

- 1-cycle penalty for a taken branch

Branch target buffer

- Cache of target addresses
- Indexed by PC when instruction fetched
  - If hit and instruction is branch predicted taken, can fetch target immediately

Agenda

- A one cycle implementation of LEGv7-M
- A pipelined implementation of LEGv7-M
- Branch prediction
- Cache memory

Principle of Locality

- Programs access a small proportion of their address space at any time
- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

https://en.wikipedia.org/wiki/Induction_variable

Example of Locality

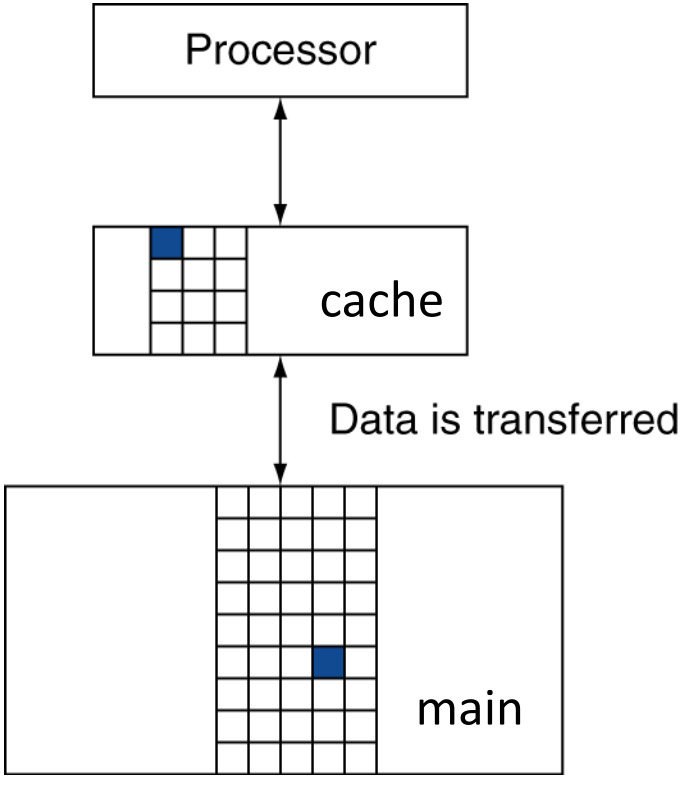

```
int A[1024];
int B[1024];
int C[1024];

    for (int i = 0; i < 1024; i++) {
        A[i] = B[i] + C[i];
    }
```

- Where is the temporal locality (if any)?
- Where is the spatial locality (if any)?

Taking Advantage of Locality

- In many modern processors the processor is (much) faster than RAM.
- Memory hierarchy
  - Copy more recently accessed (and nearby) items from main memory to smaller faster memory
  - Cache memory attached to CPU

Memory Hierarchy Levels



Block (aka line): unit of copying

- May be multiple words **Why?**

If accessed data is present in upper level

- Hit: access satisfied by upper level
  - Hit ratio: hits/accesses

If accessed data is absent

- Miss: block copied from lower level
  - Extra time taken: miss penalty
  - Miss ratio: misses/accesses
    = 1 – hit ratio
- Then accessed data supplied from upper level

STM32F411VET6

Max clock speed = 100 MHz

- No cache needed for RAM.
- Contains Adaptive real-time memory accelerator for Flash memory section 3.4.2 of RM0383
  - To release the processor full performance, the accelerator implements an instruction prefetch queue and branch cache which increases program execution speed from the 128-bit Flash memory.
  - I-code bus (instructions): cache 64 lines of 128 bits
  - D-code bus (literals): cache 8 lines of 128 bits

Advanced computer architecture

- SIMD instructions, Arm Cortex-M4
- Superscalar Wikipedia
- Out of order execution Wikipedia
- Speculative execution Wikipedia
- Simultaneous multithreading (SMT) Wikipedia
- Multi-core Wikipedia

Cyclone V FPGA (HWP01+CSC10) contains Arm Cortex-A9
which contains many of the features mentioned above

Volgende les…

Scheduling.

Aan de slag!

Aan de slag met [Opdrachten_Week_2.pdf]