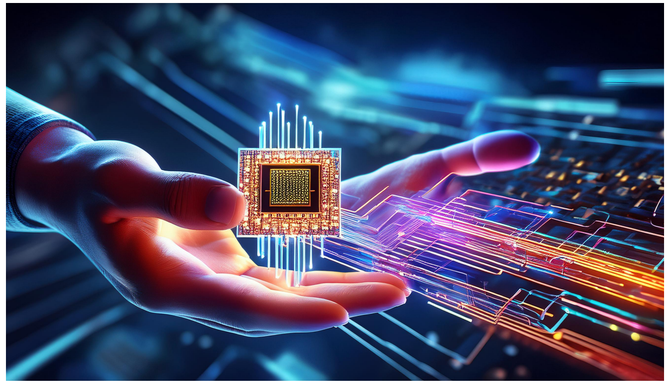
12.11.24 14:52

CSCI 250
Introduction to Computer Organisation
Lecture 4: Control Unit and Pipelines II



Jetic Gū

2024 Fall Semester (S3)

Overview

- Architecture: von Neumann
- Textbook: CO: 4.5
- Core Ideas:
    1. Pipelined Computers I
    2. ARM Thumb Instructions: Branch
    3. Lab 4 Part 1

P0
Multiple-Cycle

3 Primary Stages of CPU Execution[1]

- **Fetch**
  - The fetching of information from either the Main Memory or Register
- **Decode**
  - The interpretation of the instruction, generating the appropriate **Datapath Control**
- **Execute**
  - Actual execution of the instruction
    After which the **NS** counter in the previous example should **reset**, so do the other Sequence Control signals

Review

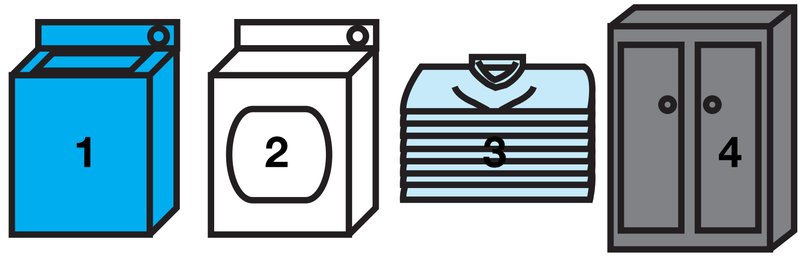
1. IBM Stretch Project, 1956-1961

CPU Pipelines

I don’t want to spend my whole life waiting

A Laundry Analogy

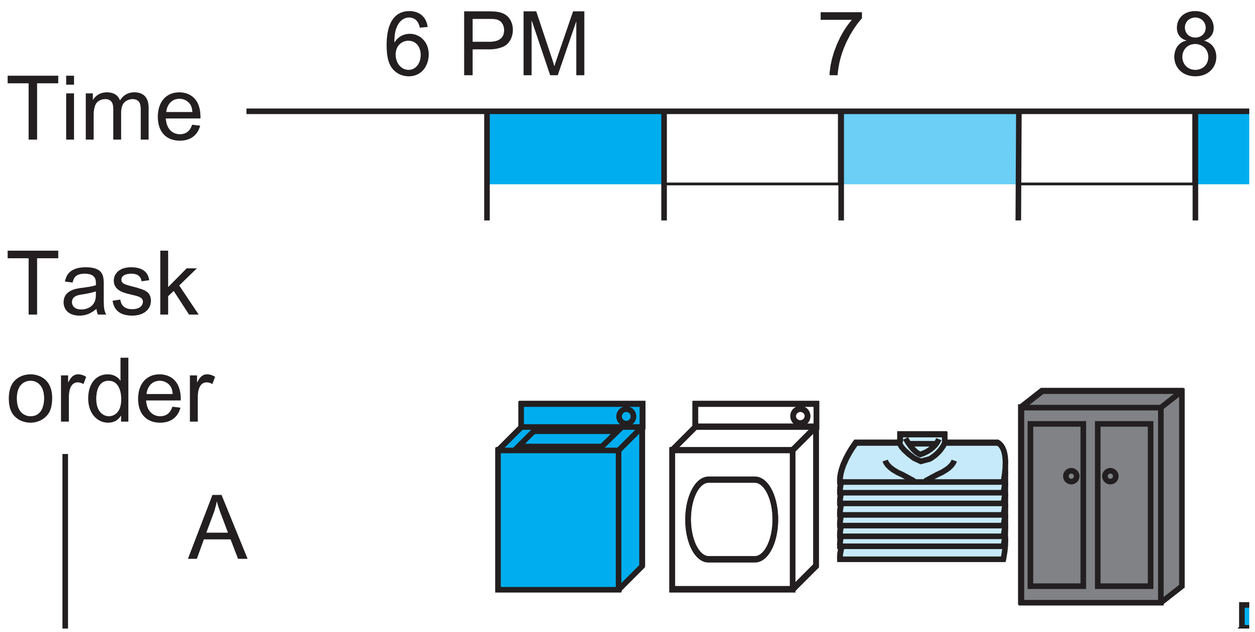
- 4 Steps of doing laundry
    1. Use the washer
    2. Use the dryer
    3. Fold the dried clothes
    4. Put the folded clothes away in the wardrobe



Example

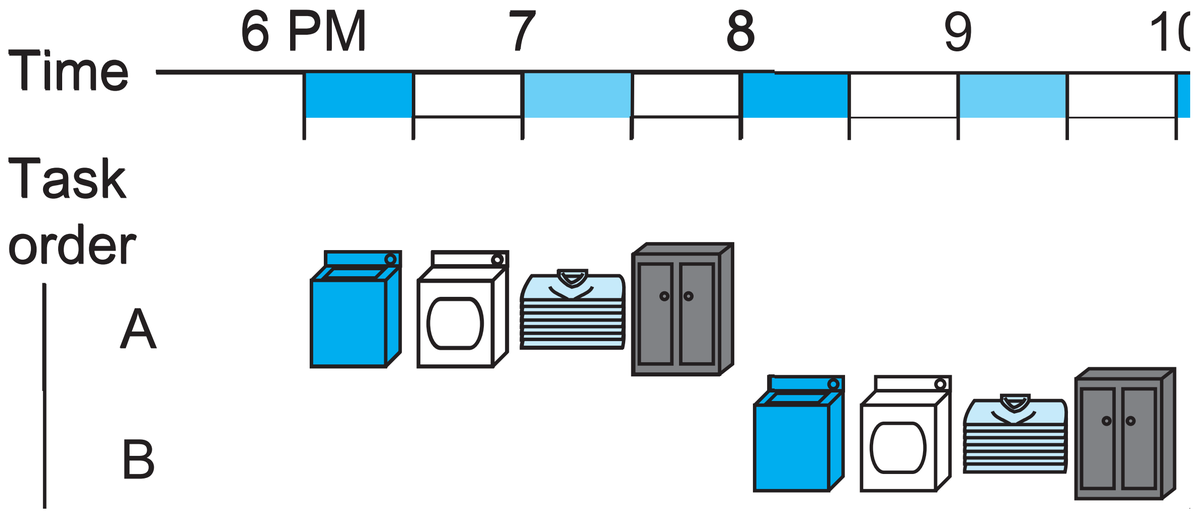
A Laundry Analogy

- Your name is A
- You do your laundry from 6pm
- It takes 2 hours for you to finish



Example

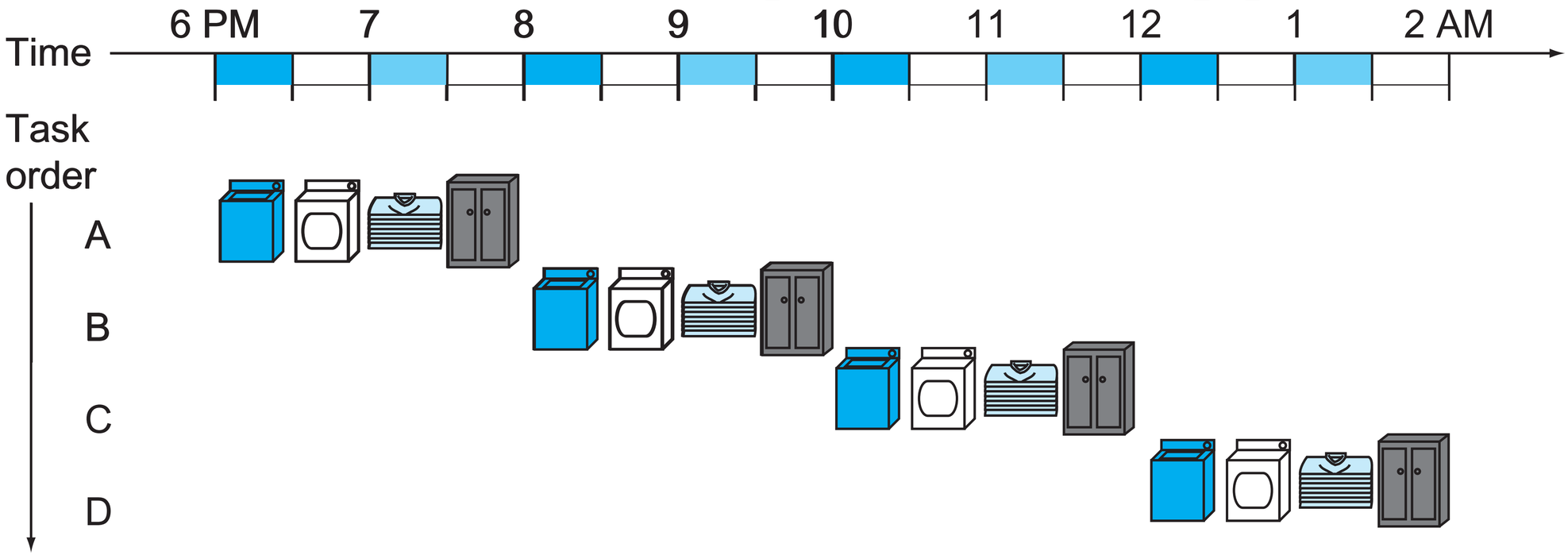
A Laundry Analogy



- Your roommate B waits for you
- B starts doing laundry from 8pm
- B finishes at 10pm

Example

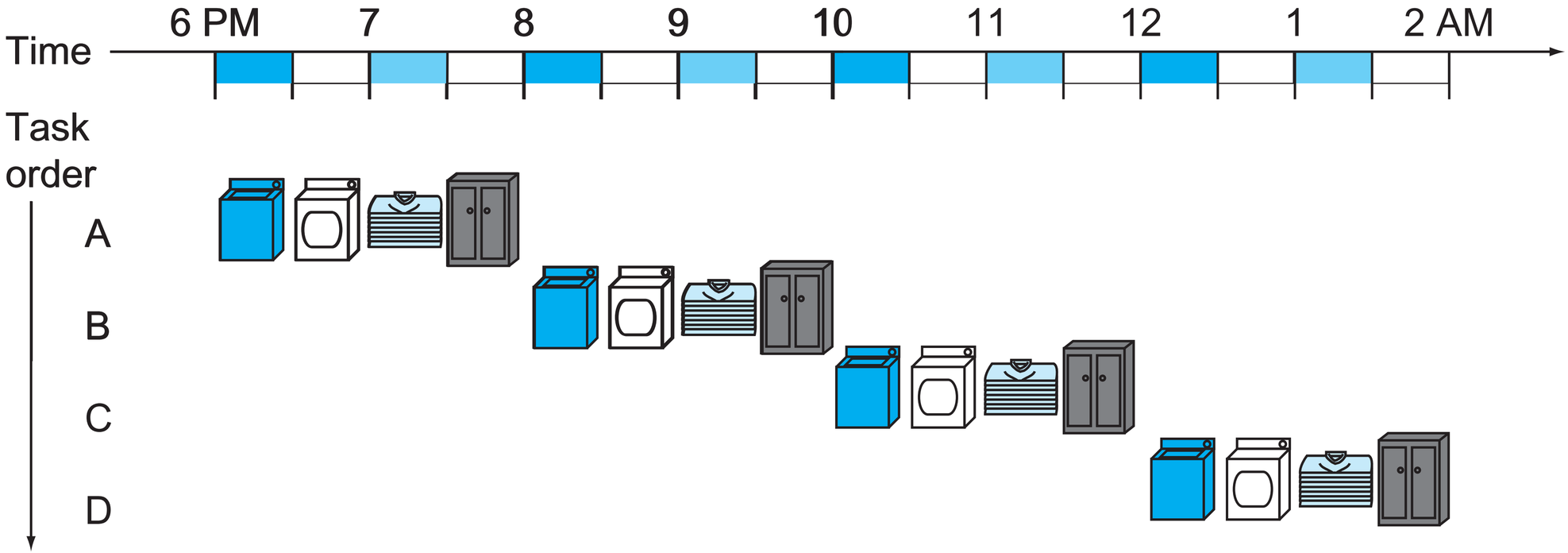
A Laundry Analogy



- If B, C, D waits for the previous person to finish organising the wardrobe before starting to use the washer, it takes a long time for all 4 people to finish their laundries

Example

A Laundry Analogy

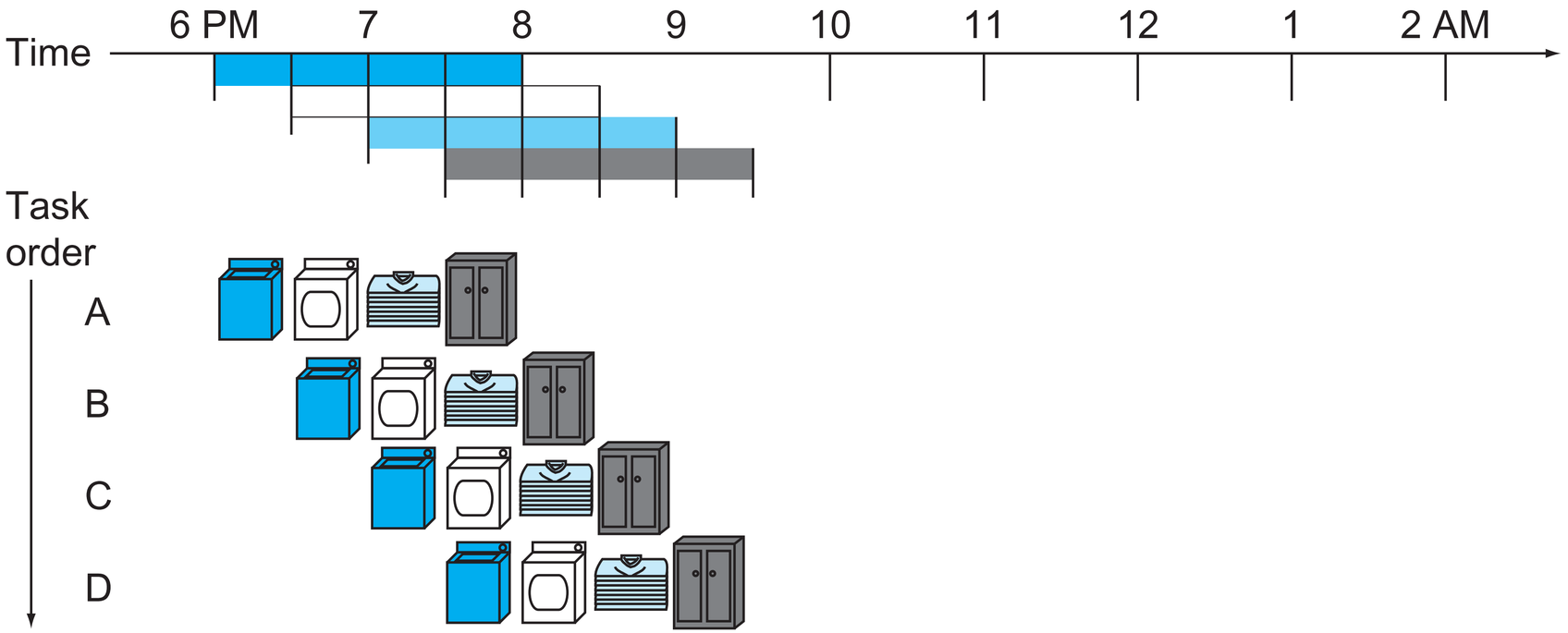


- Total duration: 6pm - 2am; 16 hours

- During which: 4 hours of washer working, 4 hours of dryer working
  75% idle, **25% efficiency (bad)**

Example

A Laundry Analogy



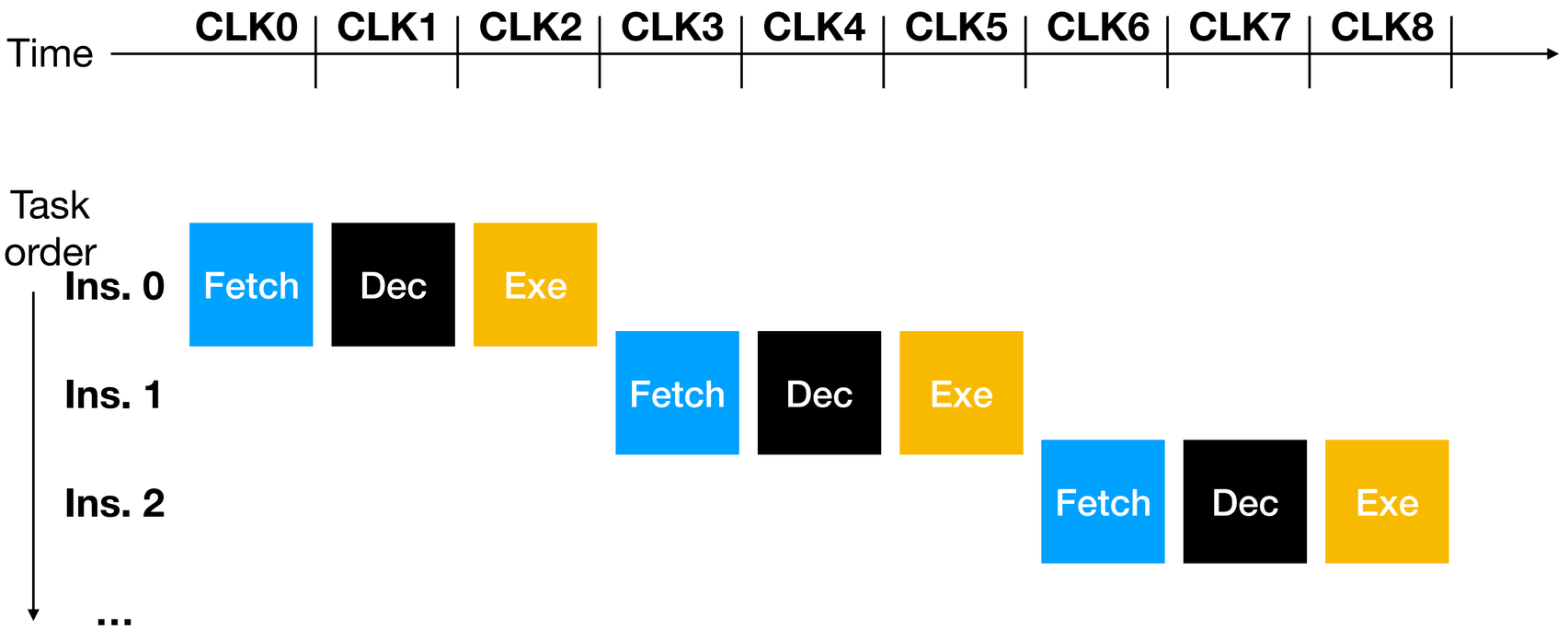
- Solution: allow the next person to use the washer, immediately after the first person ends

- Total duration: 7 hours, washer/dryer each work for 4hr; **57% efficiency**, much better

Example

P1
Pipeline

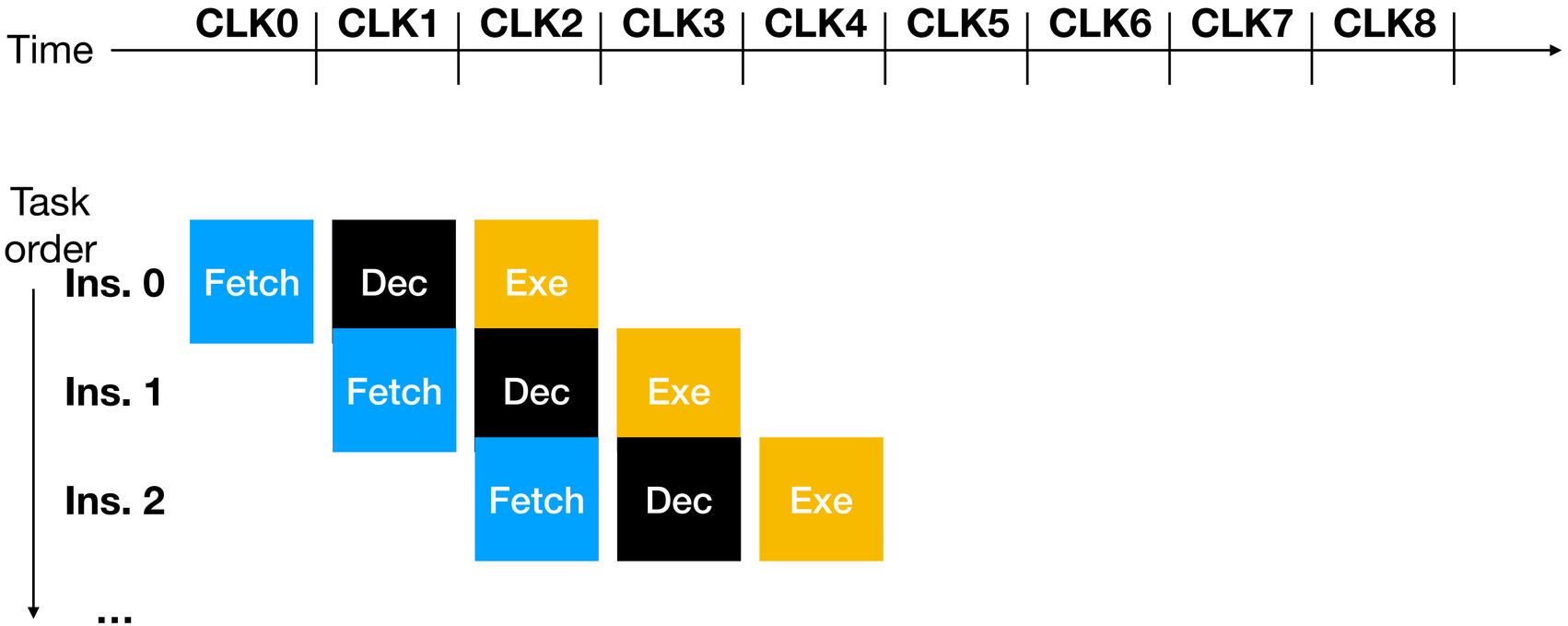
Multiple-Cycle CPU



- Say, 3 instructions needs to be executed, like we discussed last week
- Without pipeline, it’s taking quite a while

Example

Multiple-Cycle CPU



- **Pipeline** starts working on the next instruction as soon as the **current one enters the next stage**

- At peak efficiency, you can get infinitely close to 100% efficiency.
  (Think: what would be necessary?)

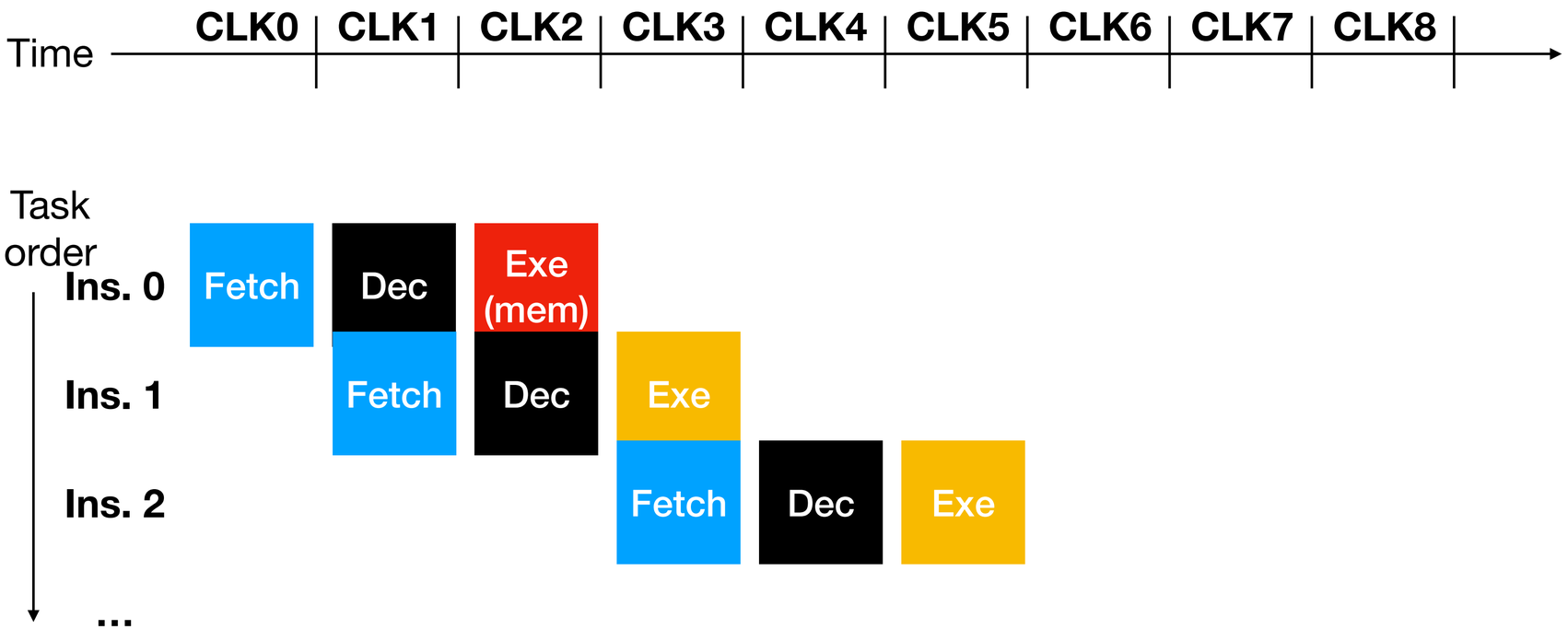
Example

Possible Issues?

- **Fetch** is a memory operation, will **involve memory**
  - In ARMv7, this involves Loading the instruction from main memory
- **Decode** doesn’t involve memory
  - In ARMv7, this involves figuring out which registers are involved in the execution of the instruction, and make them ready for the **ALU**
- **Execute** could also **involve memory** access
  - In ARMv7, ALU executes the arithmetic operation. Upon finishing, the results are written back to the register array, or **Main Memory** (**store/load** operation)

Concept

Possible Issues?



- Whenever memory operations are required in **Execute**, we cannot perform **Fetch** at the same time
  - This is called **structural hazard**: hardware cannot support the combination of instructions we want to run in the same clock cycle

Example

Example

- With the following instructions in this order, **without pipelining**, how much time is needed to execute the whole thing?
  - LDR -> LDR -> LDR -> ADD -> MOV -> SUB -> STR -> STR

Instruction	Fetch	Decode	Execute	Require Mem Access
STR Store register	4ns	0.2ps	4ns	**Yes**
LDR Load register	4ns	0.2ps	3ns	**Yes**
Add	4ns	0.2ps	0.6ps	No
Sub	4ns	0.2ps	0.6ps	No
Mov	4ns	0.2ps	0.6ps	No

Example

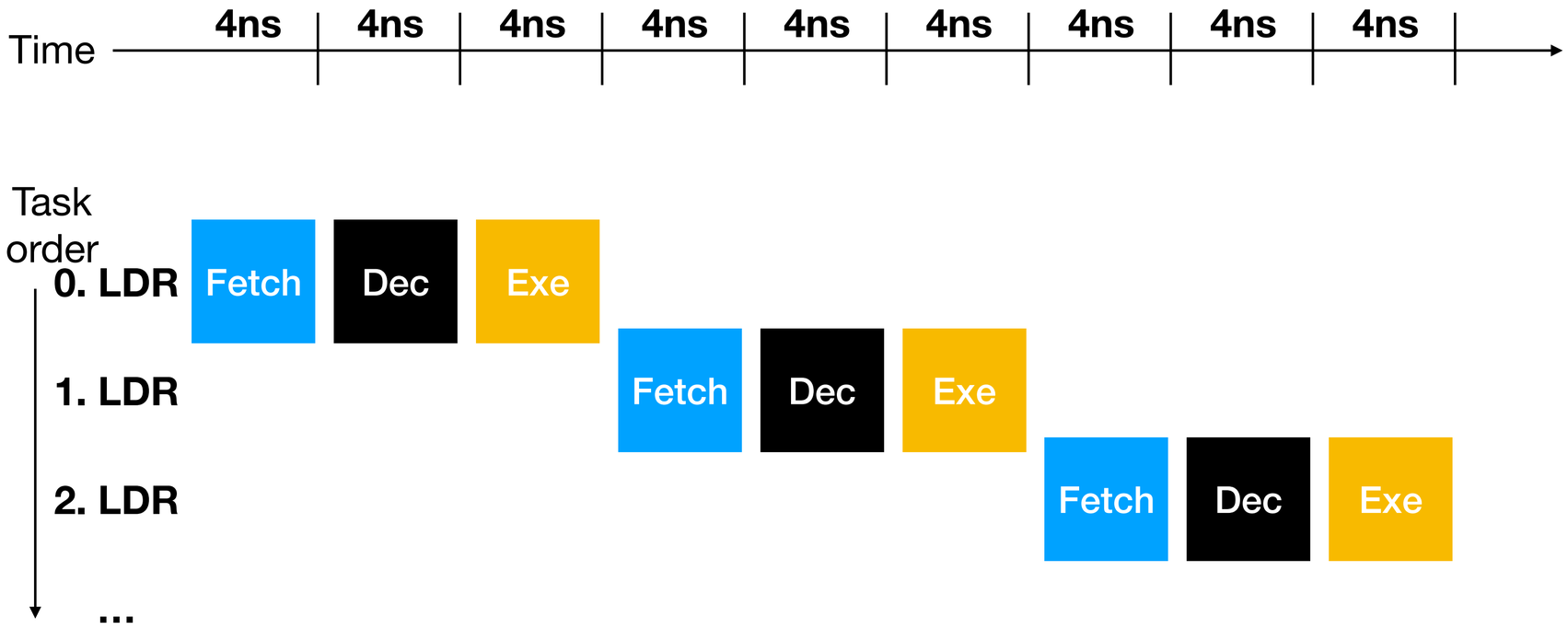
Example

- With the following instructions in this order, **with pipelining and no cache**, how much time is needed to execute the whole thing?
    - LDR -> LDR -> LDR -> ADD -> MOV -> SUB -> STR -> STR

Instruction	Fetch	Decode	Execute	Require Mem Access
STR Store register	4ns	0.2ps	4ns	**Yes**
LDR Load register	4ns	0.2ps	3ns	**Yes**
Add	4ns	0.2ps	0.6ps	No
Sub	4ns	0.2ps	0.6ps	No
Mov	4ns	0.2ps	0.6ps	No

Example

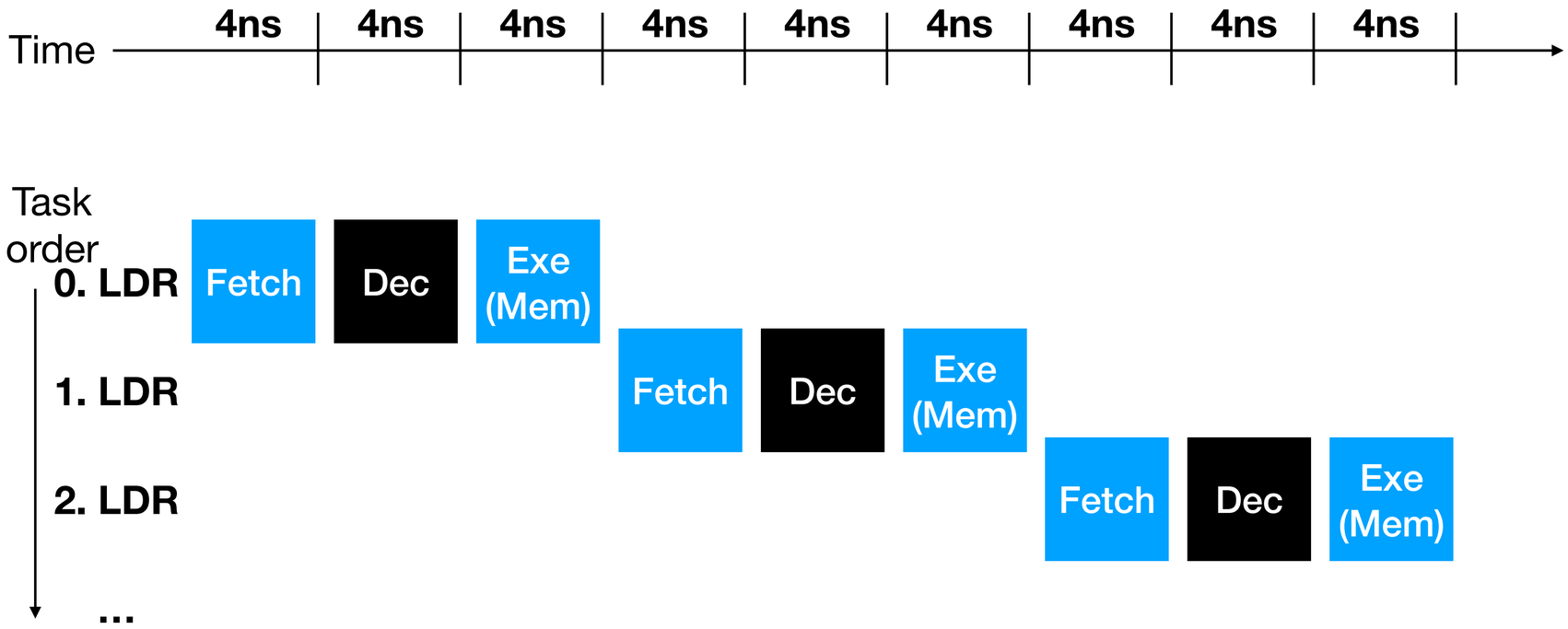
Example



- The maximum amount of time for any stage is: **4ns** (memory read/write)
- So execution time is measured in increments of **4ns**

Example

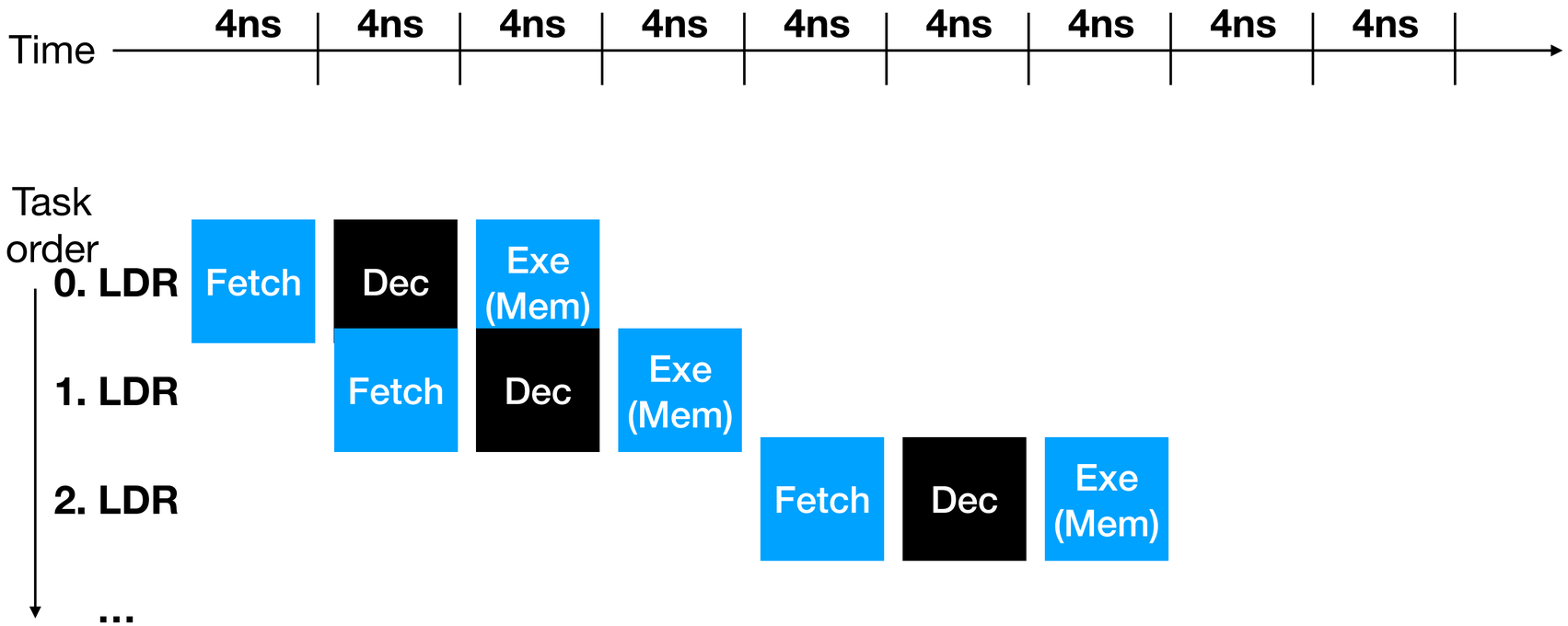
Example



- Exe stages with memory operations cannot overlap with **Fetch**, decode however is fine

Example

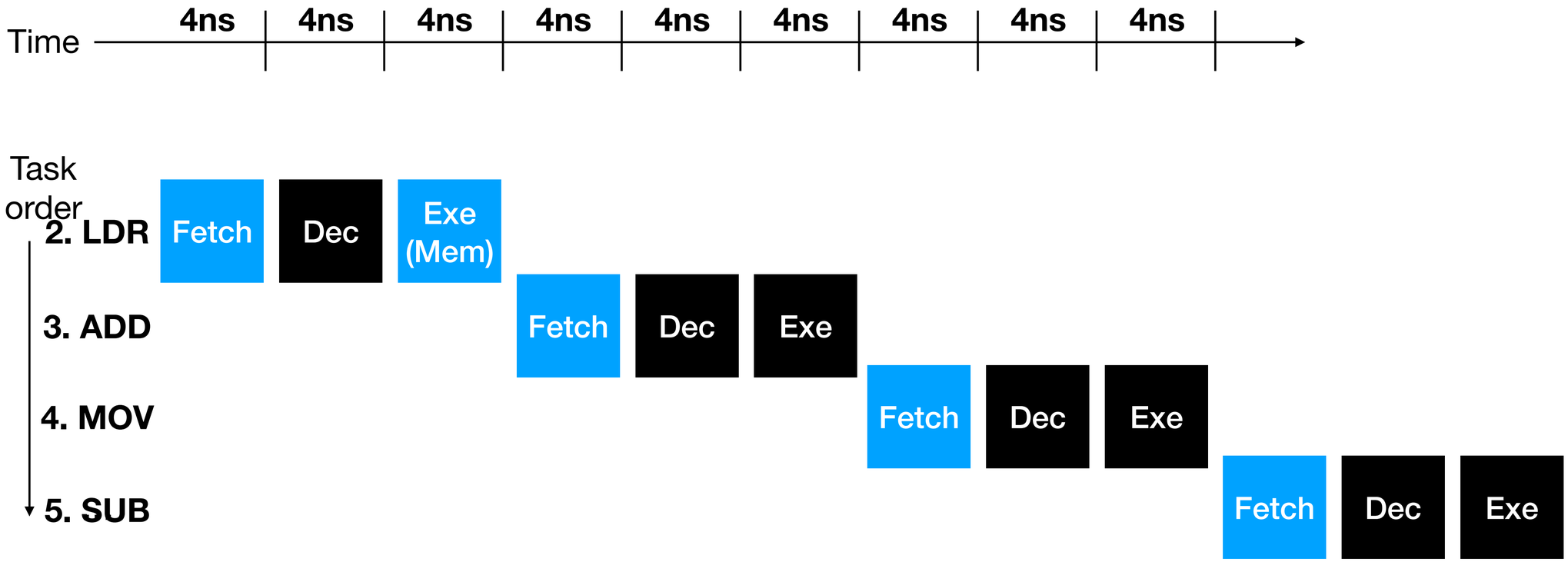
Example



- Exe stages with memory operations cannot overlap with **Fetch**, decode however is fine (time reduction: 1 x 4ns = 4 ns)

Example

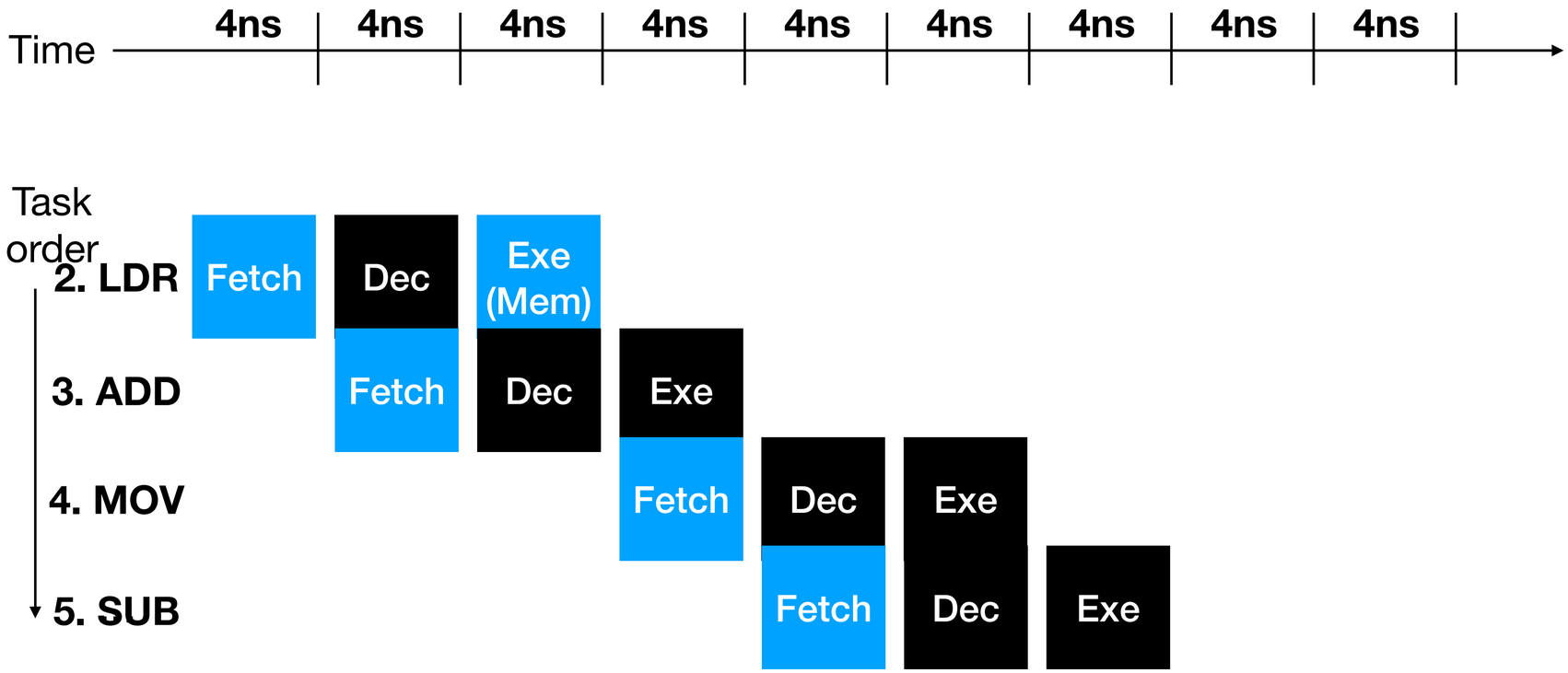
Example



- The next operations, without overlapping memory operations, can be organised as such

Example

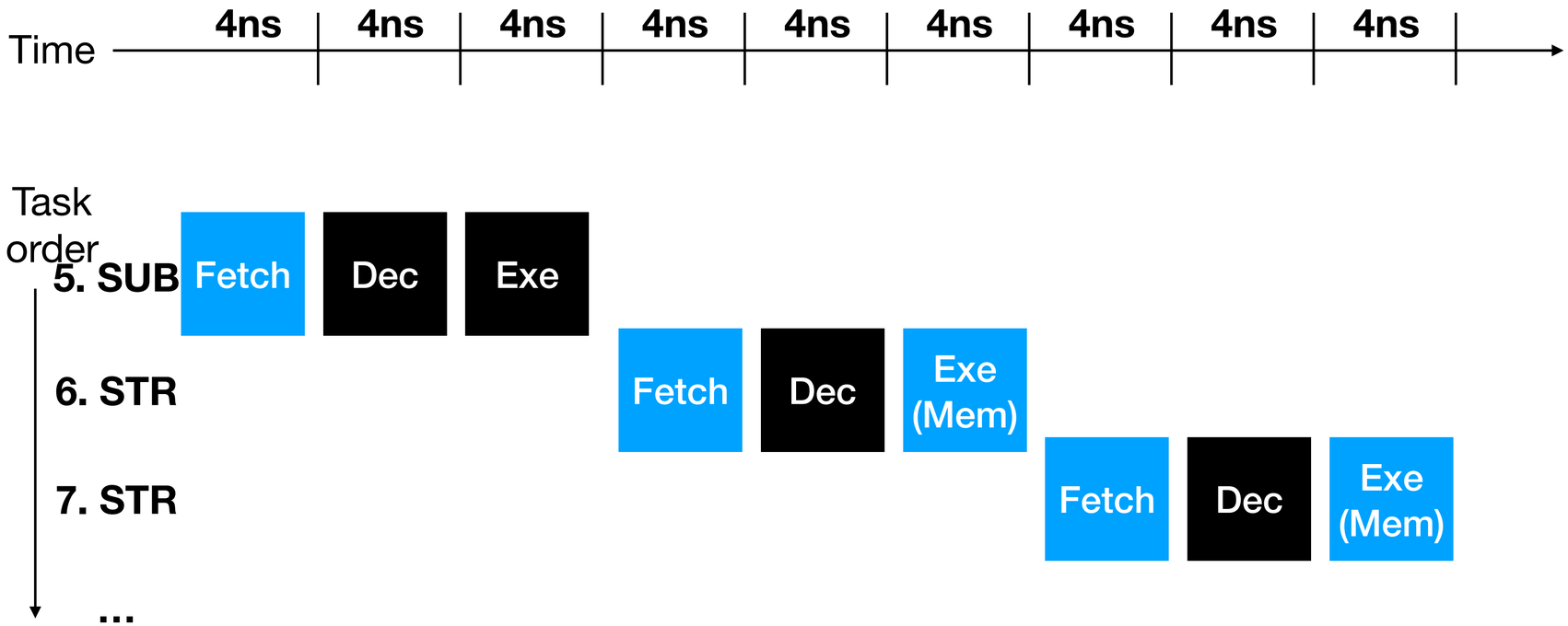
Example



- The next operations, without overlapping memory operations, can be organised as such (time reduction: 5 x 4ns = 20 ns, total: 24ns)

Example

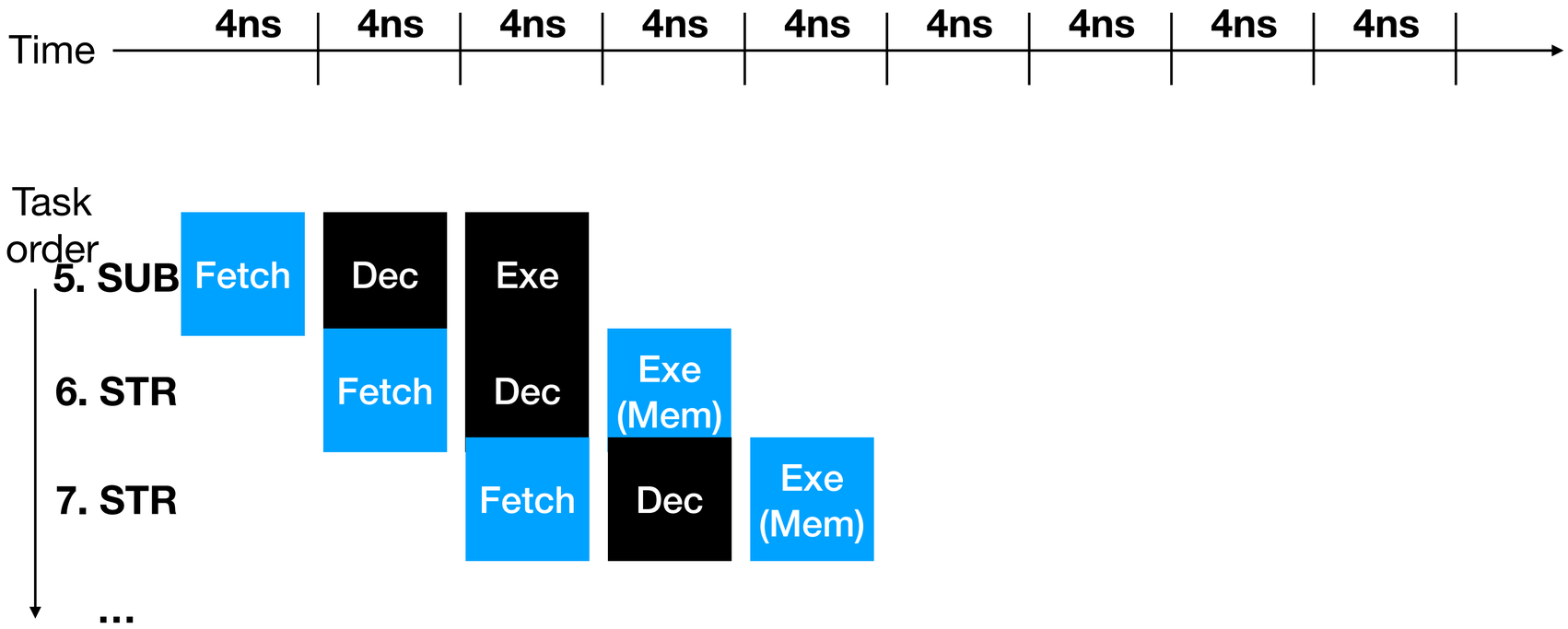
Example



- The next two STR operations needs MEM, so we have to adjust a bit

Example

Example



- The next two STR operations needs MEM, so we have to adjust a bit
  Final time reduction: 4 x 4ns = 16 ns, total savings: 40ns

- Without pipelining: 8 x 3 x 4ns = **96 ns**; With pipelining: **56 ns**;

Example

P1
Pipeline

What is the most important aspect of pipelining?

- What is the slowest operation?
- How do we optimise our pipeline so it can run the fastest?
- How do we know our pipeline is good enough?
- What instructions could render pipelines ineffective?

Example

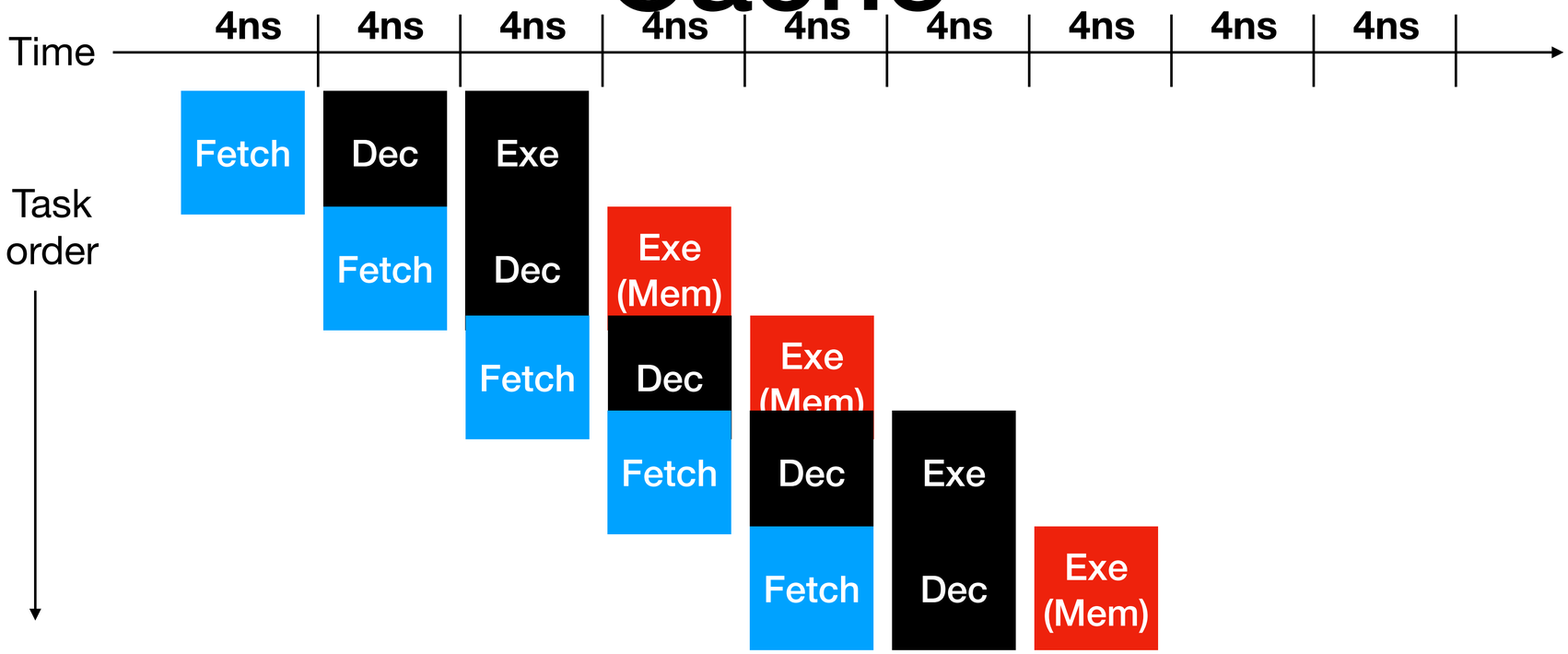
Cache

- What makes Cache vital
  - Instructions are always **Cached**
  - Instructions can be **Cached** separately from data cache
    - Meaning: modern computers with separate instruction cache and data cache, can actually allow simultaneous **Fetch** and **Execution** regardless of whether you need memory access in **Execution**!
    - Does it resolve the issue with **Branching**? Why?

Example

P1
Pipeline

Separate Instruction/Data Cache



- **Fetch** will always go to the Instruction Cache, which can be done concurrently with **Execution** memory access with Data Cache

- Instructions are not supposed to change on-the-fly, and memory writes in **Execution** is only done to Data Cache, will not affect Instruction Cache. This simplifies our design and further increases efficiency

Example

Stages of ARM CPUs

- ARM largely follows the Fetch/Decode/Execution idea, with
  - **ARMv7**: exactly these 3 stages
  - **ARMv9**: 5 stages, but each stage is much more efficient than ARMv7, so the overall efficiency is actually higher
  - **ARMv10**: 6 stage, but each stage is much more efficient than ARMv7/v9, so the overall efficiency is almost double that of ARMv7
  - **ARMv11**: 8 stages, fetch for example is divided into two; even faster than v10
- Why does having more stages give faster execution?
  Because of **Pipelining**.

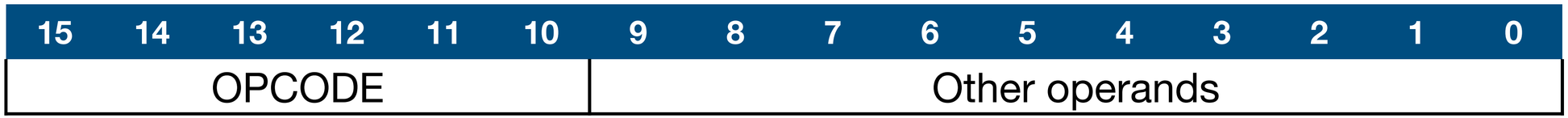
Review

1. https://developer.arm.com/documentation/ddi0333/h/introduction/pipeline-stages

ARM Branching

P2
ARM B(X)

ARM 16bit Thumb Instructions



Opcode	Instruction Encoding	
`00xxxx`	Register/Immediate: Arithmetic Operations	AU
`010000`	Register: Logical Operations	LU
`010001`	Special Register data instructions*	**BX**
`01001x`	Memory Load: from Literal Pool (PC with Offset)	MEM

`0101xx`
`011xxx`
`100xxx`
	Memory Load/Store (Single address)	MEM

`1010XX`	Relative Address calculation*	
`1011xx`	Misc*	
`1100xx`	Memory Load/Store (Blocks)*	
`1101xx`	Conditional branch: `if`-triggered subroutine/`goto`	**B**
`11100x`	Unconditional branch: `jump`	**B**

1. https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Thumb-Instruction-Set-Encoding/16-bit-Thumb-instruction-encoding

P2
ARM B(X)

Branching / Jump is a simple idea

- **Jump / Unconditional Branching**
  - We are changing the value of PC
- **Conditional Branching**
  - We are changing the value of PC, when certain condition is met

Review

P2
ARM B(X)

Branch and Exchange (BX[1])

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	0	Rm			0	0	0	0

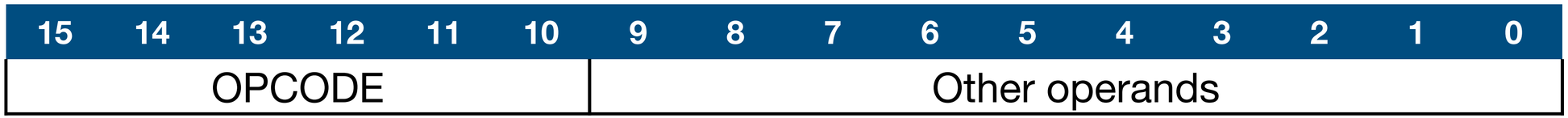
- **Special data instructions and branch and exchange**
  - This is the only instruction in this category that we care about
- **BX**
  - Take the value from Register `Rm`, give it to PC (`R7`)
  - Why you should use BX instead of MOV when assigning new value to PC?

1. I changed the spec here to reflect our lower number of registers than ARM32

Technical

P2
ARM B(X)

ARM 16bit Thumb Instructions



Opcode	Instruction Encoding	
`00xxxx`	Register/Immediate: Arithmetic Operations	AU
`010000`	Register: Logical Operations	LU
`010001`	Special Register data instructions*	**BX √**
`01001x`	Memory Load: from Literal Pool (PC with Offset)	MEM

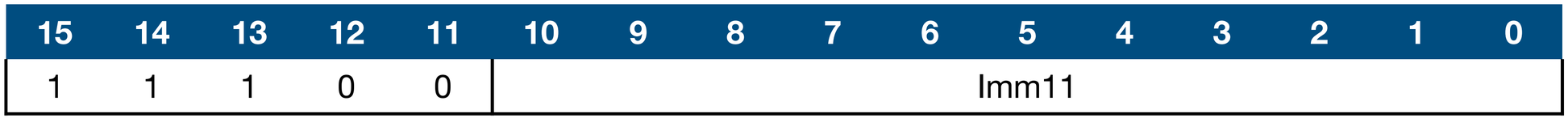
`0101xx`
`011xxx`
`100xxx`
	Memory Load/Store (Single address)	MEM

`1010XX`	Relative Address calculation*	
`1011xx`	Misc*	
`1100xx`	Memory Load/Store (Blocks)*	
`1101xx`	Conditional branch: `if`-triggered subroutine/`goto`	**B**
`11100x`	Unconditional branch: `jump`	**B**

1. https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Thumb-Instruction-Set-Encoding/16-bit-Thumb-instruction-encoding

P2
ARM B(X)

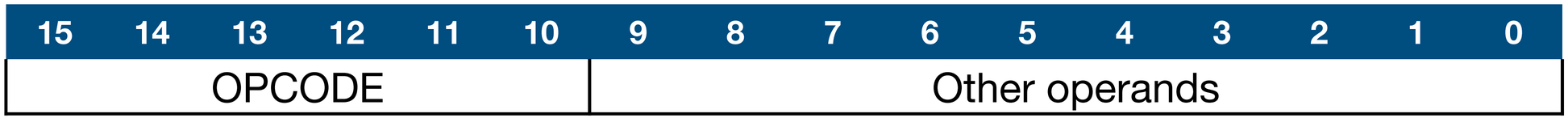
Unconditional branch: `jump`



- **Unconditional Jump**
  - Format: `B Imm11`
  - `PC` receives value from `Imm11`

P2
ARM B(X)

ARM 16bit Thumb Instructions



Opcode	Instruction Encoding	
`00xxxx`	Register/Immediate: Arithmetic Operations	AU
`010000`	Register: Logical Operations	LU
`010001`	Special Register data instructions*	**BX √**
`01001x`	Memory Load: from Literal Pool (PC with Offset)	MEM

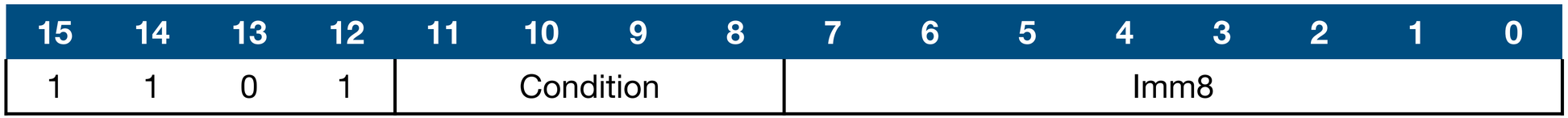
`0101xx`
`011xxx`
`100xxx`
	Memory Load/Store (Single address)	MEM

`1010XX`	Relative Address calculation*	
`1011xx`	Misc*	
`1100xx`	Memory Load/Store (Blocks)*	
`1101xx`	Conditional branch: `if`-triggered subroutine/`goto`	**B**
`11100x`	Unconditional branch: `jump`	**B√**

1. https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Thumb-Instruction-Set-Encoding/16-bit-Thumb-instruction-encoding

P2
ARM B(X)

Conditional Branch: `B`



- **Conditional branch, and Supervisor Call**
  - We won't discuss supervisor call here, this is like a **System Call** to the OS
- **Format**: `B.cond Imm8`
- **Condition:** `cond`
  - The condition here is a 4 digit code `cond` that takes a look at the result of a preceding `CMP` (compare) operation or arithmetic operation

P2
ARM B(X)

CMP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	0	Rm			Rn		

- **CMP**
  - **Format**: `CMP Rn Rm`
  - Compares the values in register `Rm` and `Rn`
  - The results are stored in a special Flag Register, that is NOT a GPR
    - Specification for the Flag Register:

Flag	Z=1	Z=0	C=1	C=0	N=1	N=0	V
Operations	CMP	CMP	CMP	CMP	CMP	CMP	ADD/SUB
Meaning	Rn == Rm	Rn != Rm	Rn >= Rm	Rn < Rm	Rn < 0	Rn >= 0	Overflow

P2
ARM B(X)

Conditional Branch: B

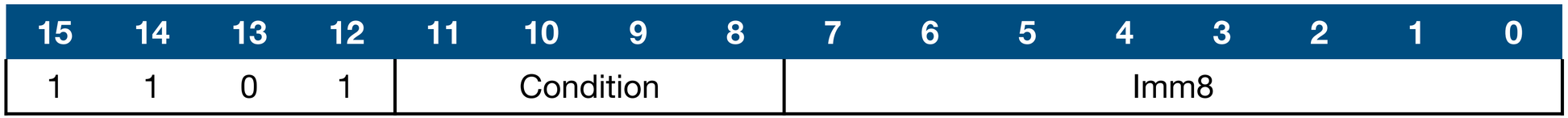
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Condition				Imm8							

ConditionCode	Label	Flag Register content
0000	EQ	Z == 1 (Equal)
0001	NE	Z == 0 (Not Equal)
0010	CS	C == 1 (Greater or equal)
0011	CC	C == 0 (Lesser)
0100	MI	N == 1 (Negative)
0101	PL	N == 0 (Non-negative)
0110	VS	V == 1 (There was overflow)
0111	VC	V == 0 (There wasn’t overflow)
1000	HI	C and notZ (Greater than)
1001	LS	notC or Z (Lessor or equal)
1010	GE	(N and V) or (notN and notV) (greater or equal)
1011	LT	N xor V (less than)
1100	GT	notZ and GE (greater than)
1101	LE	Z or LT (less than)
1110	AL	Always (Condition is always met)
1111	NV	Always (Condition is never met)

1. http://www.righto.com/2016/01/conditional-instructions-in-arm1.html

P2
ARM B(X)

Conditional Branch: B



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Condition				Imm8							

- **Format**: `B.cond Imm8`
- **Condition:** `cond`
  - With the condition code, check the flag register to see if the condition is met
    - If met, PC receives new value from `Imm8`

P2
ARM B(X)

Think

- How does Memory operations affect the pipeline?
- How does branching affect the pipeline?

Future

P3
Lab

Lab 4 part 1

Building CMP and Condition Code checker

P3
Lab

Building a CMP component

- Build a `CMP` component, with 3 internal flip-flops. The component must be done using circuit diagrams
- Input: 16bit `Rn`, 16bit `Rm`, 4bit condition code `cond`, 1bit enable `E`, 1bit `CLK`. Let's assume `V` is always 0, so no need to worry about carry.
- Output: `F`, dependent on `cond` and internal states `N`, `C`, `Z`, `V`.
- When `E <= 0`
  - Internal flip-flops' values do not change.
- When `E <= 1`
  - Perform comparison, update `N`, `C`, `Z` at `CLK`

Technical

P3
Lab

Think

- Is there an elegant way to include the CMP in your AL?

Technical