Pipelining

CSU34021 - Computer Architecture II

History / Background

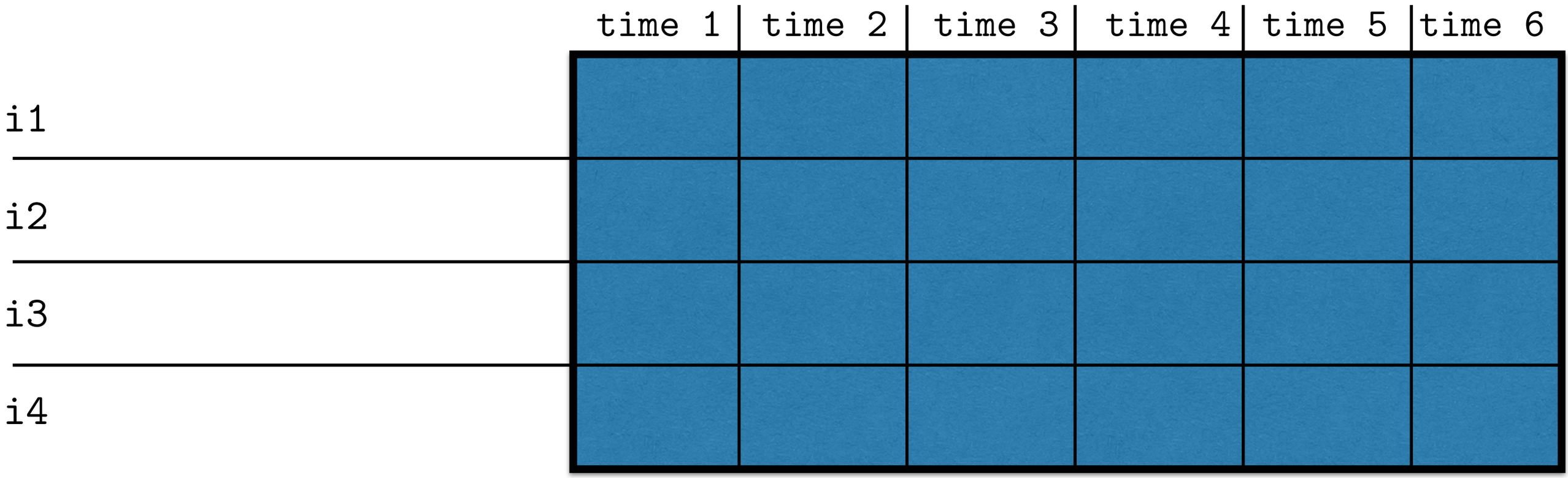
- Pipelining is a key technique used by modern CPUs for increasing throughput.
- First (static) pipelined microprocessors in mid 1980s (before that pipelining mostly for supercomputers and mainframes).
- Dynamic pipelining schemes to become widespread in desktop computers in the 1990s.
- RISC architectures intentionally designed for ease of implementing pipelining [overlap of fetch-execute of RISC-1 an example of two-stage pipeline].

Pipeline Design Principle

- Break each instruction into a series of small steps that can be executed in parallel [think of a car assembly line].

- Each instruction takes the same amount of time [actually on average a bit higher] but overall throughput increases!

Five ‘small’ steps, no pipelining:

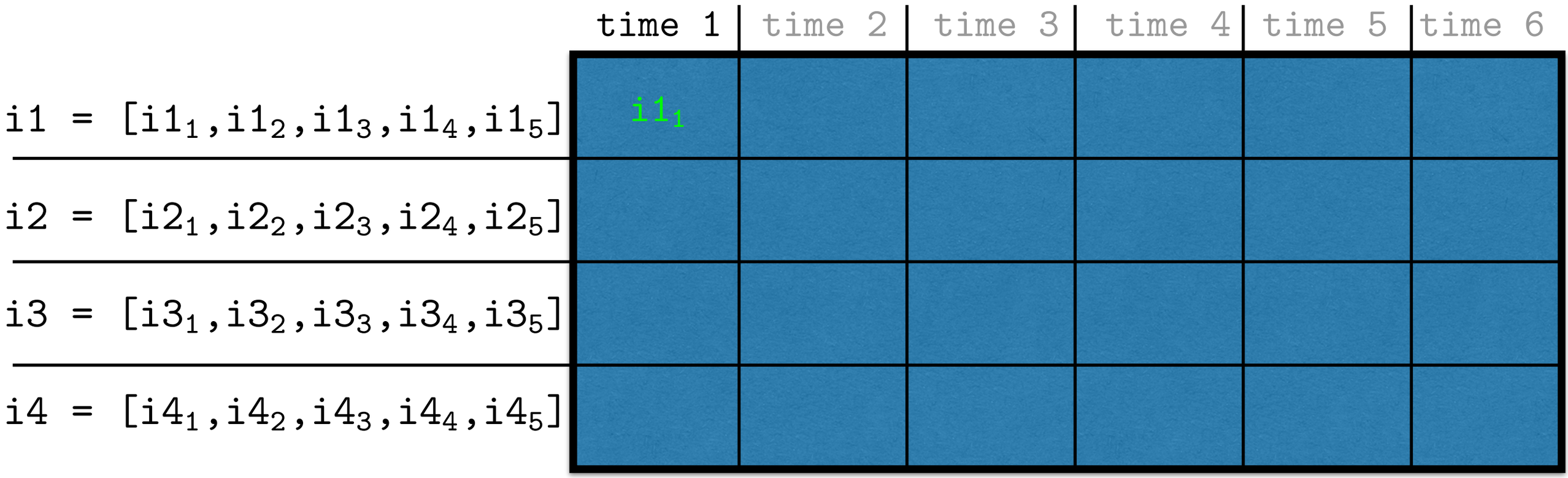


Pipeline Design Principle

- Break each instruction into a series of small steps that can be executed in parallel [think of a car assembly line].

- Each instruction takes the same amount of time [actually on average a bit higher] but overall throughput increases!

Five ‘small’ steps, no pipelining:

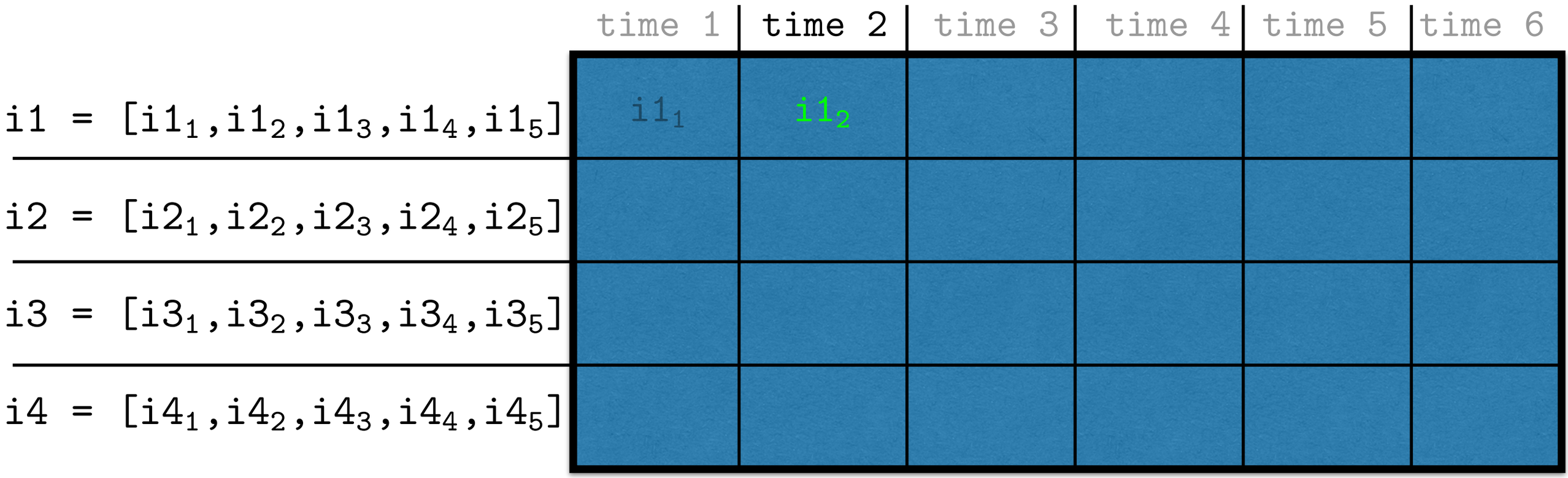


Pipeline Design Principle

- Break each instruction into a series of small steps that can be executed in parallel [think of a car assembly line].

- Each instruction takes the same amount of time [actually on average a bit higher] but overall throughput increases!

Five ‘small’ steps, no pipelining:

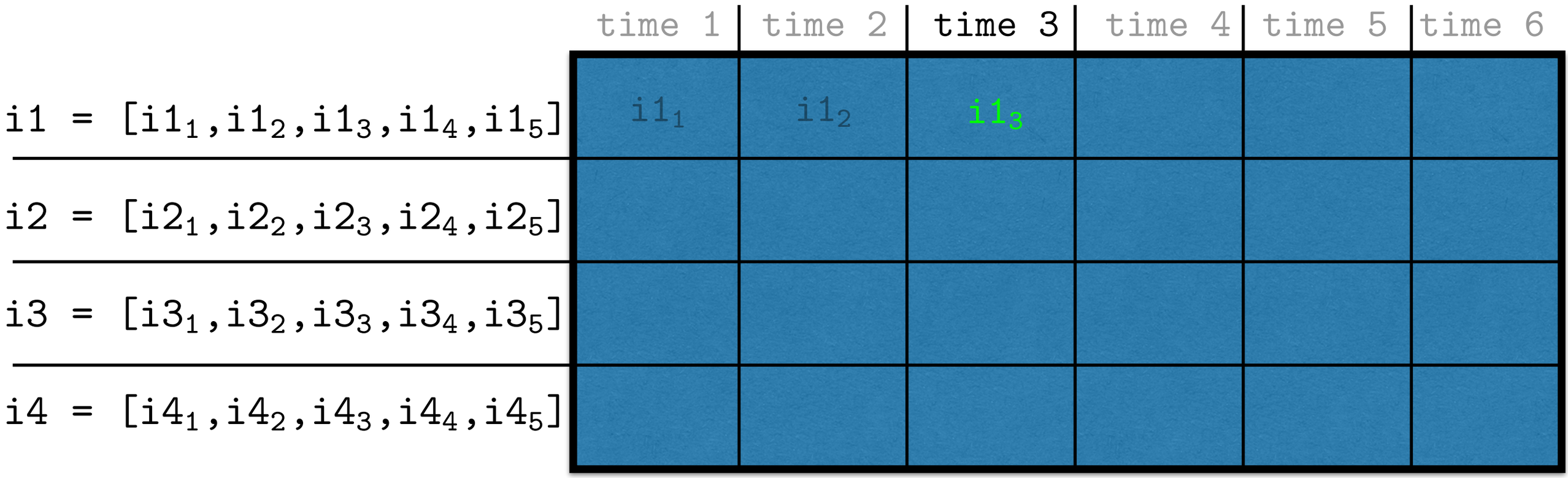


Pipeline Design Principle

- Break each instruction into a series of small steps that can be executed in parallel [think of a car assembly line].

- Each instruction takes the same amount of time [actually on average a bit higher] but overall throughput increases!

Five ‘small’ steps, no pipelining:

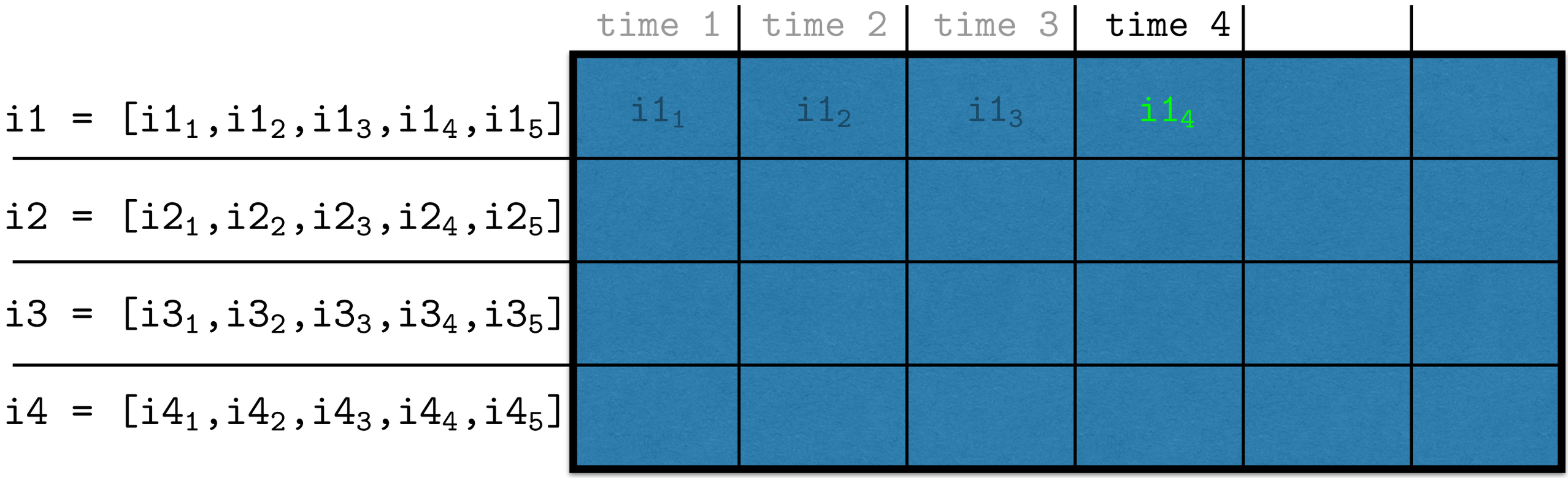


Pipeline Design Principle

- Break each instruction into a series of small steps that can be executed in parallel [think of a car assembly line].

- Each instruction takes the same amount of time [actually on average a bit higher] but overall throughput increases!

Five ‘small’ steps, no pipelining:

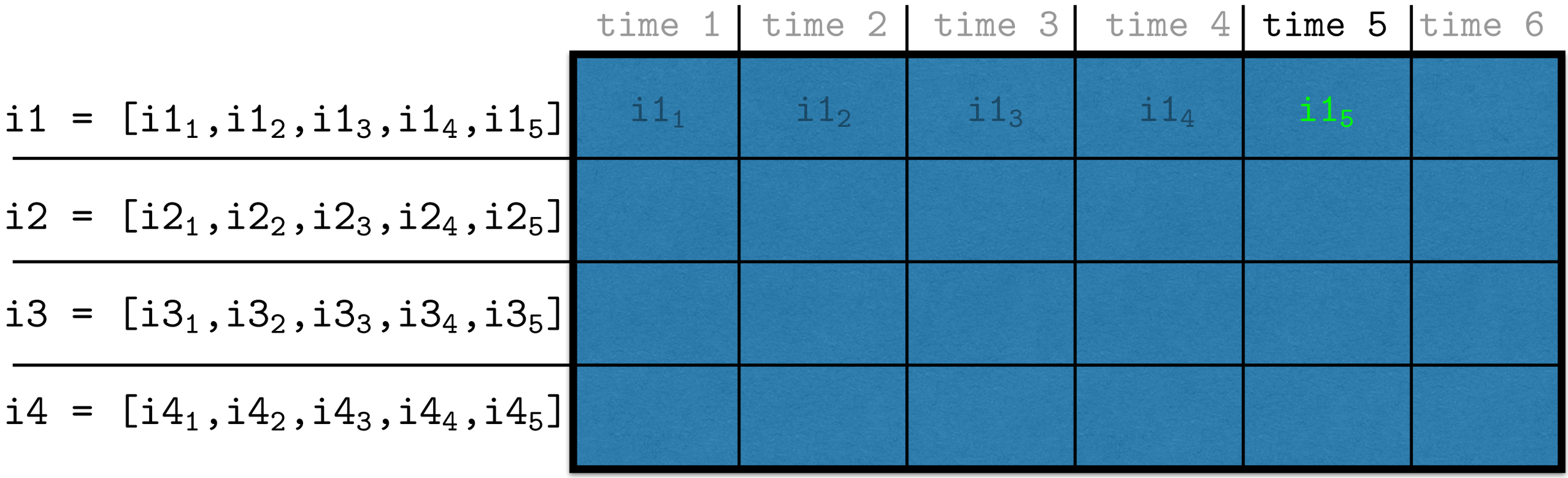


Pipeline Design Principle

- Break each instruction into a series of small steps that can be executed in parallel [think of a car assembly line].

- Each instruction takes the same amount of time [actually on average a bit higher] but overall throughput increases!

Five 'small' steps, no pipelining:

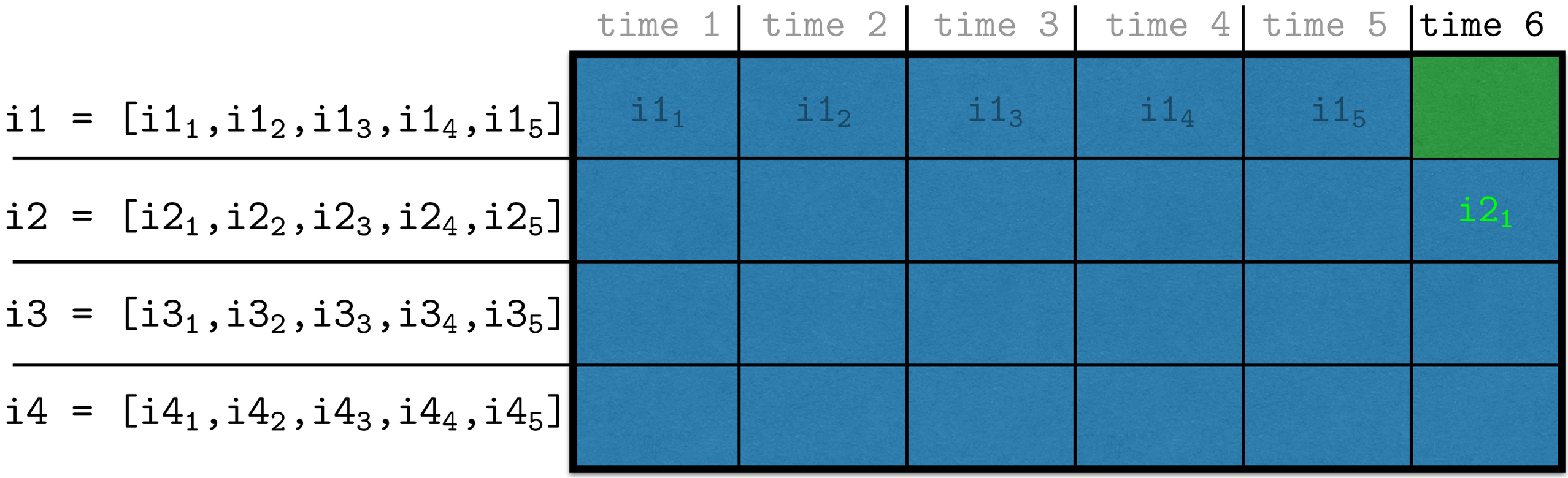


Pipeline Design Principle

- Break each instruction into a series of small steps that can be executed in parallel [think of a car assembly line].

- Each instruction takes the same amount of time [actually on average a bit higher] but overall throughput increases!

Five ‘small’ steps with pipelining:

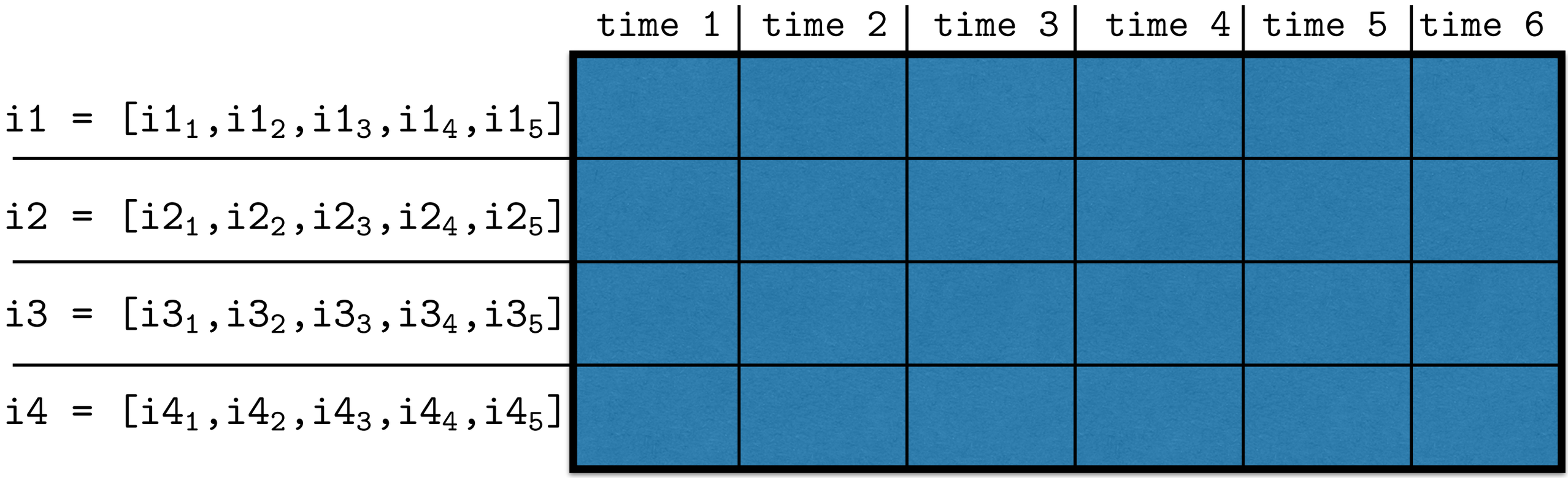


Pipeline Design Principle

- Break each instruction into a series of small steps that can be executed in parallel [think of a car assembly line].

- Each instruction takes the same amount of time [actually on average a bit higher] but overall throughput increases!

Five ‘small’ steps with pipelining:

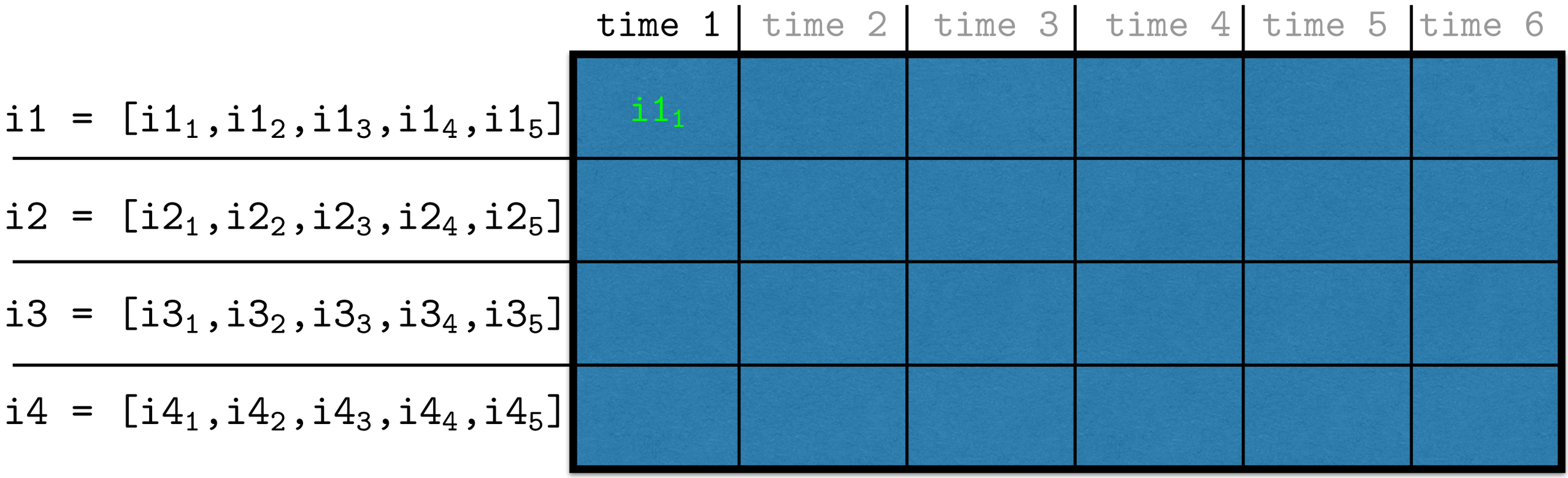


Pipeline Design Principle

- Break each instruction into a series of small steps that can be executed in parallel [think of a car assembly line].

- Each instruction takes the same amount of time [actually on average a bit higher] but overall throughput increases!

Five ‘small’ steps with pipelining:

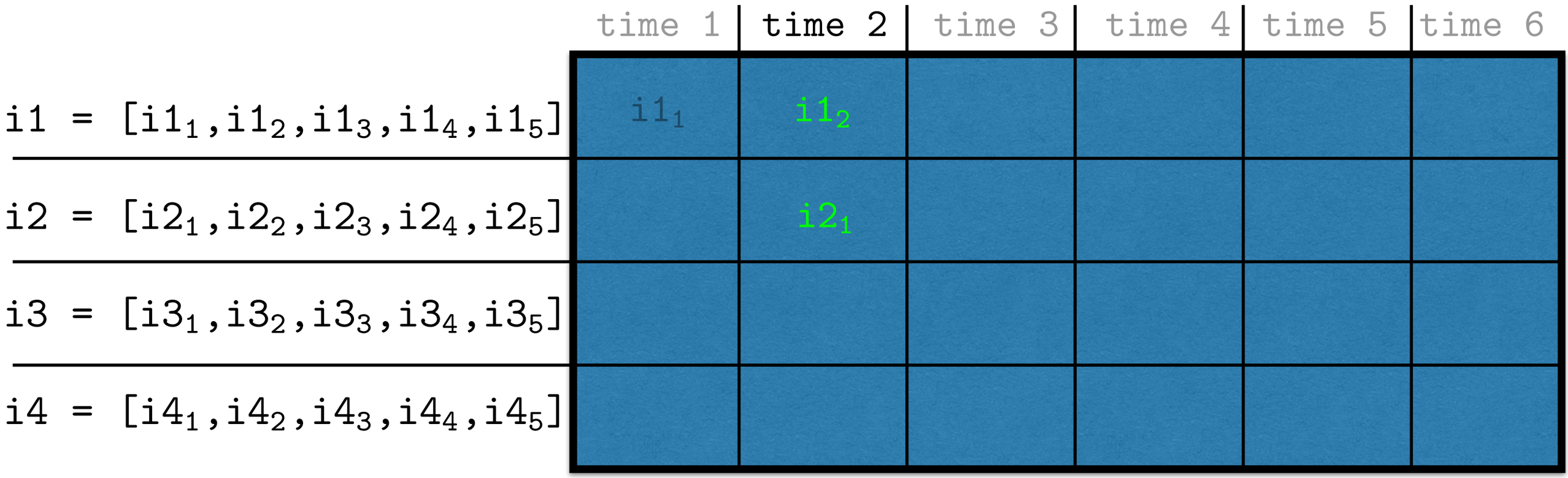


Pipeline Design Principle

- Break each instruction into a series of small steps that can be executed in parallel [think of a car assembly line].

- Each instruction takes the same amount of time [actually on average a bit higher] but overall throughput increases!

Five ‘small’ steps with pipelining:

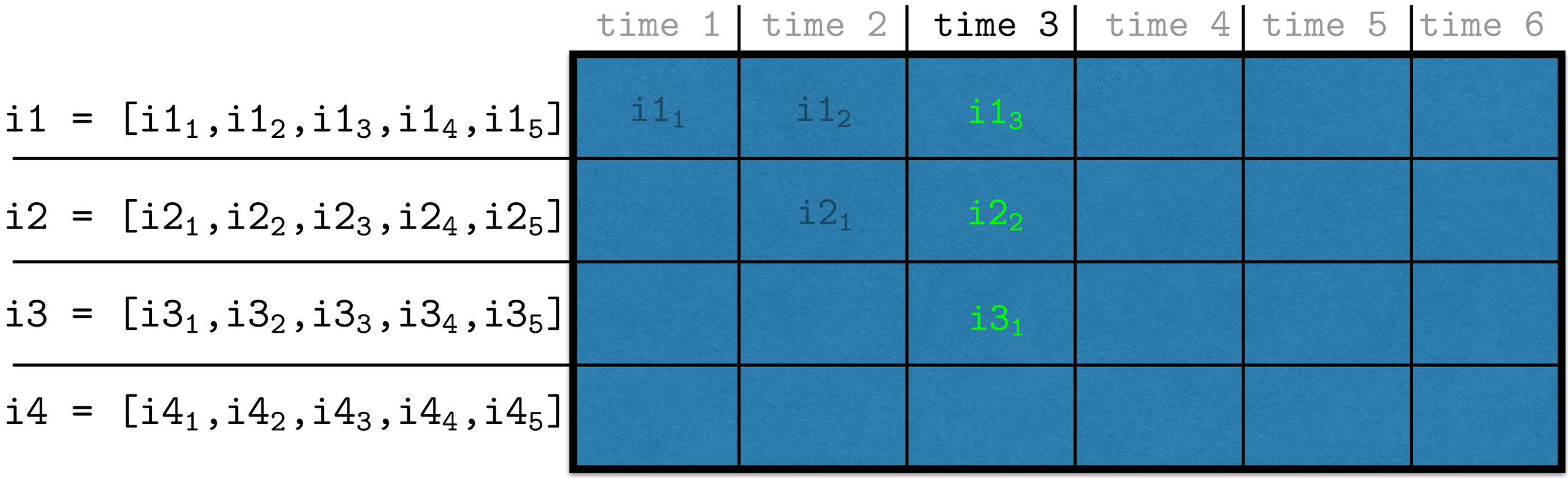


Pipeline Design Principle

- Break each instruction into a series of small steps that can be executed in parallel [think of a car assembly line].

- Each instruction takes the same amount of time [actually on average a bit higher] but overall throughput increases!

Five 'small' steps with pipelining:

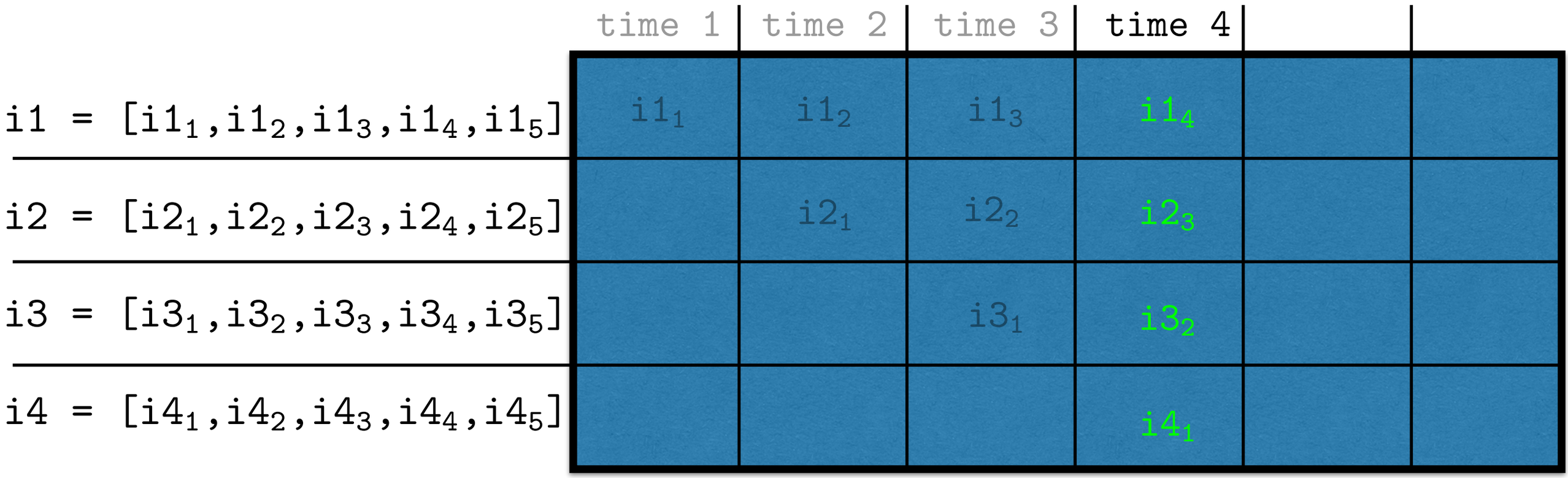


Pipeline Design Principle

- Break each instruction into a series of small steps that can be executed in parallel [think of a car assembly line].

- Each instruction takes the same amount of time [actually on average a bit higher] but overall throughput increases!

Five ‘small’ steps with pipelining:



Pipeline Design Principle

- Break each instruction into a series of small steps that can be executed in parallel [think of a car assembly line].

- Each instruction takes the same amount of time [actually on average a bit higher] but overall throughput increases!

Five ‘small’ steps with pipelining:

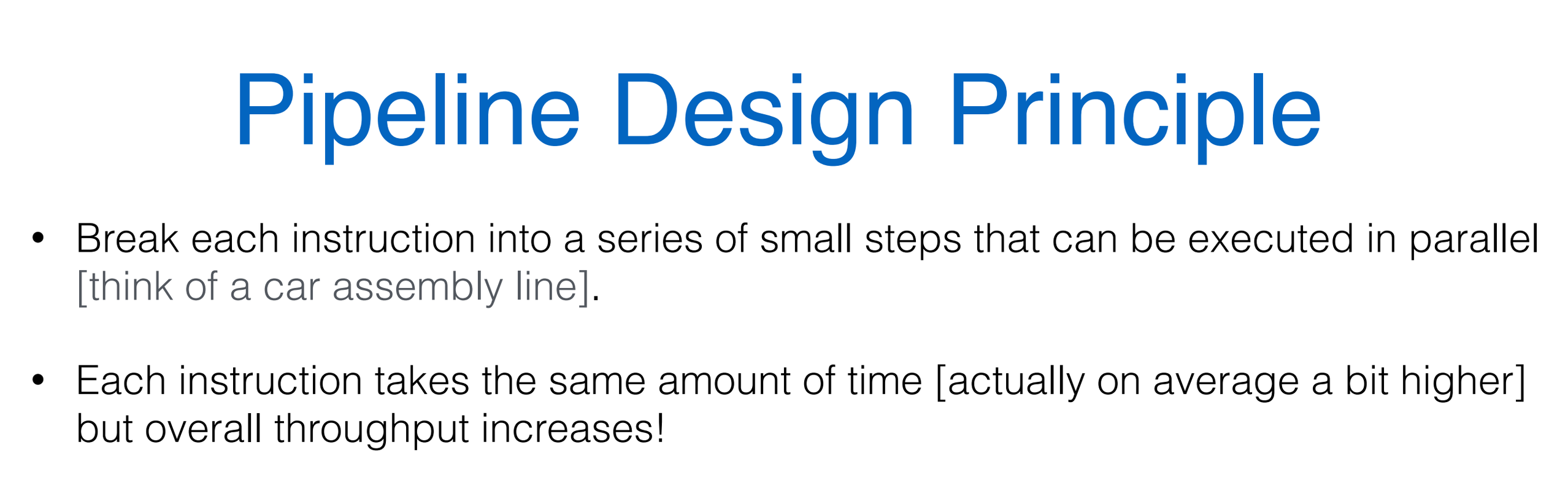
	time 1	time 2	time 3	time 4	time 5	time 6
$i1 = [i1_1, i1_2, i1_3, i1_4, i1_5]$	$i1_1$	$i1_2$	$i1_3$	$i1_4$	$i1_5$	
$i2 = [i2_1, i2_2, i2_3, i2_4, i2_5]$		$i2_1$	$i2_2$	$i2_3$	$i2_4$	
$i3 = [i3_1, i3_2, i3_3, i3_4, i3_5]$			$i3_1$	$i3_2$	$i3_3$	
$i4 = [i4_1, i4_2, i4_3, i4_4, i4_5]$				$i4_1$	$i4_2$	

Pipeline Design Principle

- Break each instruction into a series of small steps that can be executed in parallel [think of a car assembly line].

- Each instruction takes the same amount of time [actually on average a bit higher] but overall throughput increases!

Five 'small' steps with pipelining:



Pipelining

- The slowest sub-instruction sets the timing of the pipeline
- How to divide each instruction in a series of sub-instructions?
- We have to make sure that the sub-instructions can overlap!
- How are pipeline implemented?
- We will look at the DLX/MIPS pipeline.

DLX/MIPS pipeline

- DLX is a simple RISC design (not actual implementation) similar to MIPS.
- It has a five-stage pipeline. Each instruction is divided onto:

```
IF - Instruction Fetch

 ID - Instruction Decode and Register Fetch [operands]

 EX - Execution / Effective Address Calculation

MEM - Memory access

 WB - Write Back
```

DLX/MIPS Pipeline

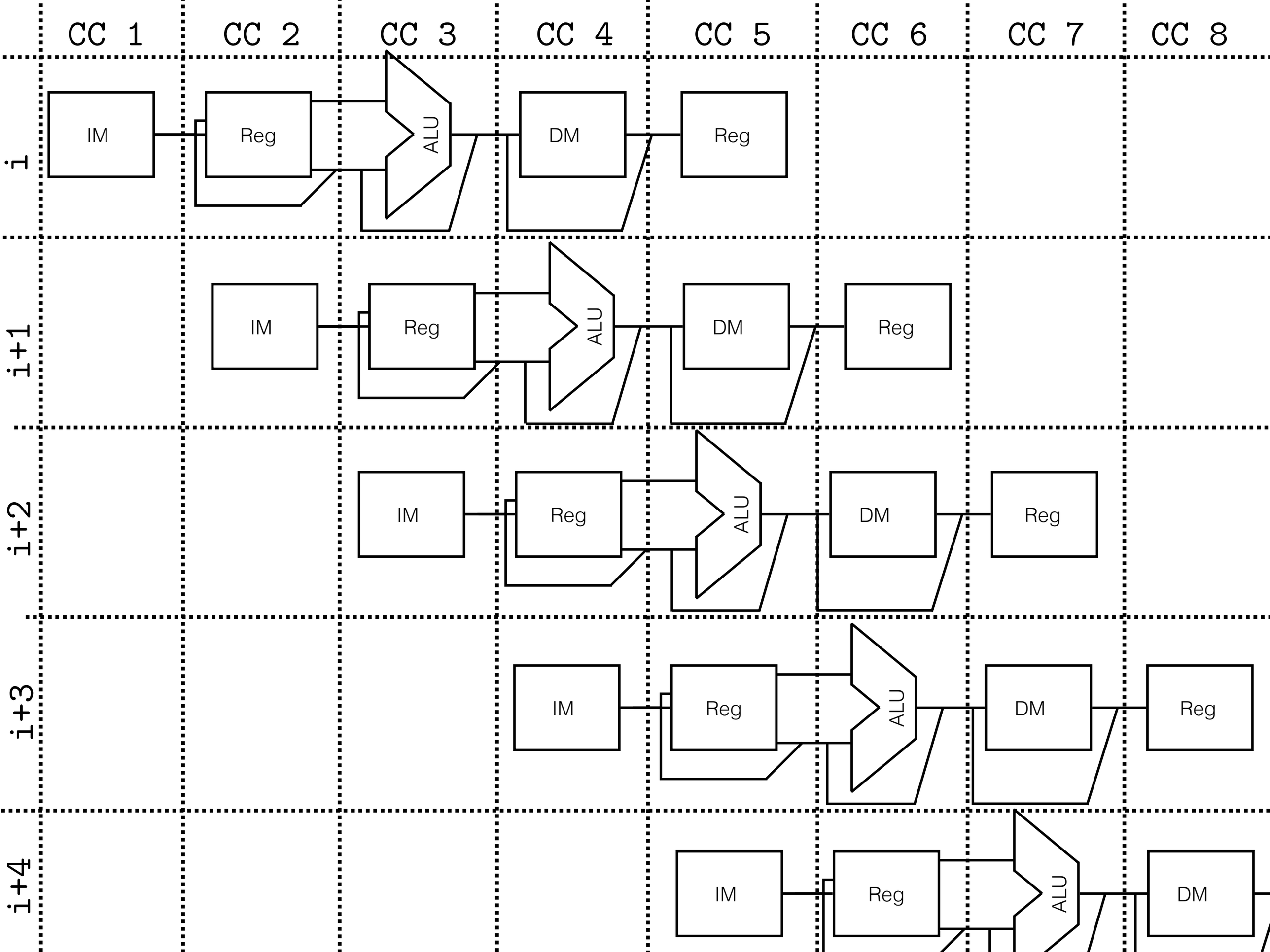
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8
i	IF	ID	EX	MEM	WB			
i+1		IF	ID	EX	MEM	WB		
i+2			IF	ID	EX	MEM	WB	
i+3				IF	ID	EX	MEM	WB
					IF	ID	EX	MEM

- In order for pipeline to function properly we need to make sure that all the 5 steps can be run simultaneously, at least generally…

DLX/MIPS Pipeline

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8
i	IF	ID	EX	MEM	WB			
i+1		IF	ID	EX	MEM	WB		
i+2			IF	ID	EX	MEM	WB	
i+3				IF	ID	EX	MEM	WB
					IF	ID	EX	MEM

- In order for pipeline to function properly we need to make sure that all the 5 steps can be run simultaneously, at least generally…
- For example: Memory access required by MEM and IF simultaneously. Solved by having two caches – one for data (MEM) and one for instruction (IF).


CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
i
i+1
i+2
i+3
i+4
IM
Reg
ALU
DM
Reg

Hazards

- Hazards: Situations in pipelining which the following instruction cannot be executed during is designated clock cycle.
- Different Type of Hazards: Structural, Data and Control hazards.
- Major hurdle of pipeline implementations and major risk for decreased performances of pipeline.

Structural Hazards

- Arise from resource conflicts: hardware cannot support all combinations of instructions simultaneously, by design.

- Example: if we didn't have two separate caches for data and instruction, every memory access would generate a stall or bubble.

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8
i=load	IF	ID	EX	MEM	WB			
i+1		IF	ID	EX	MEM	WB		
i+2			IF	ID	EX	MEM	WB	
i+3				stall	IF	ID	EX	MEM
i+4						IF	ID	EX

Example: Load/Store Structural Hazard

- Each stall reduces the performance of the pipeline. What would be the effect of the load/store structural hazard on performance?

- Clocks Per Instruction: CPI. (Ideal case: CPI=1, sub-optimal: CPI>1).

- Assume: frequency of memory reference = 40%.

- Assume no other stalls take place. Then:

$$CPI = CPI_{ideal} + 0.4*1 = 1.4$$

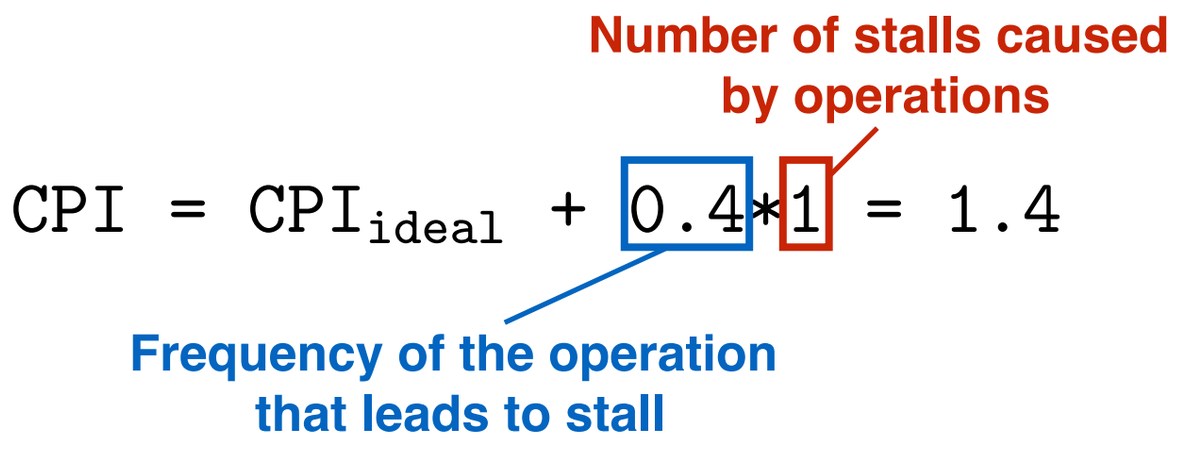
Example: Load/Store Structural Hazard

- Each stall reduces the performance of the pipeline. What would be the effect of the load/store structural hazard on performance?

- Clocks Per Instruction: CPI. (Ideal case: CPI≈1, sub-optimal: CPI>1).

- Assume: frequency of memory reference = 40%.

- Assume no other stalls take place. Then:



Example: Load/Store Hazard, quantitative analysis

- Pipelining the load/store hazard requires as to use separate instruction for cache and data. Assume the version with two caches has a clock speed 1.05 times slower than the one with the structural hazard. Which one is the fastest overall?

$$avg\_time_{pipd} = CPI_{pipd} * clock\_cycle\_time_{pipd} = 1 * 1.05 * clock\_cycle\_time_{un\text{-}pipd}$$

$$avg\_time_{un\text{-}pipd} = CPI_{un\text{-}pipd} * clock\_cycle\_time_{un\text{-}pipd} = 1.4 * clock\_cycle\_time_{un\text{-}pipd}$$

- Clearly the version without hazard is faster. The speed up obtained is:

$$speed\text{-}up = avg\_time_{un\text{-}pipd} / avg\_time_{pipd} = (CPI_{un\text{-}pipd} * clock\_cicle\_time_{un\text{-}pipd}) / (CPI_{pipd} * clock\_cicle\_time_{pipd}) = 1.33$$

Data Hazards

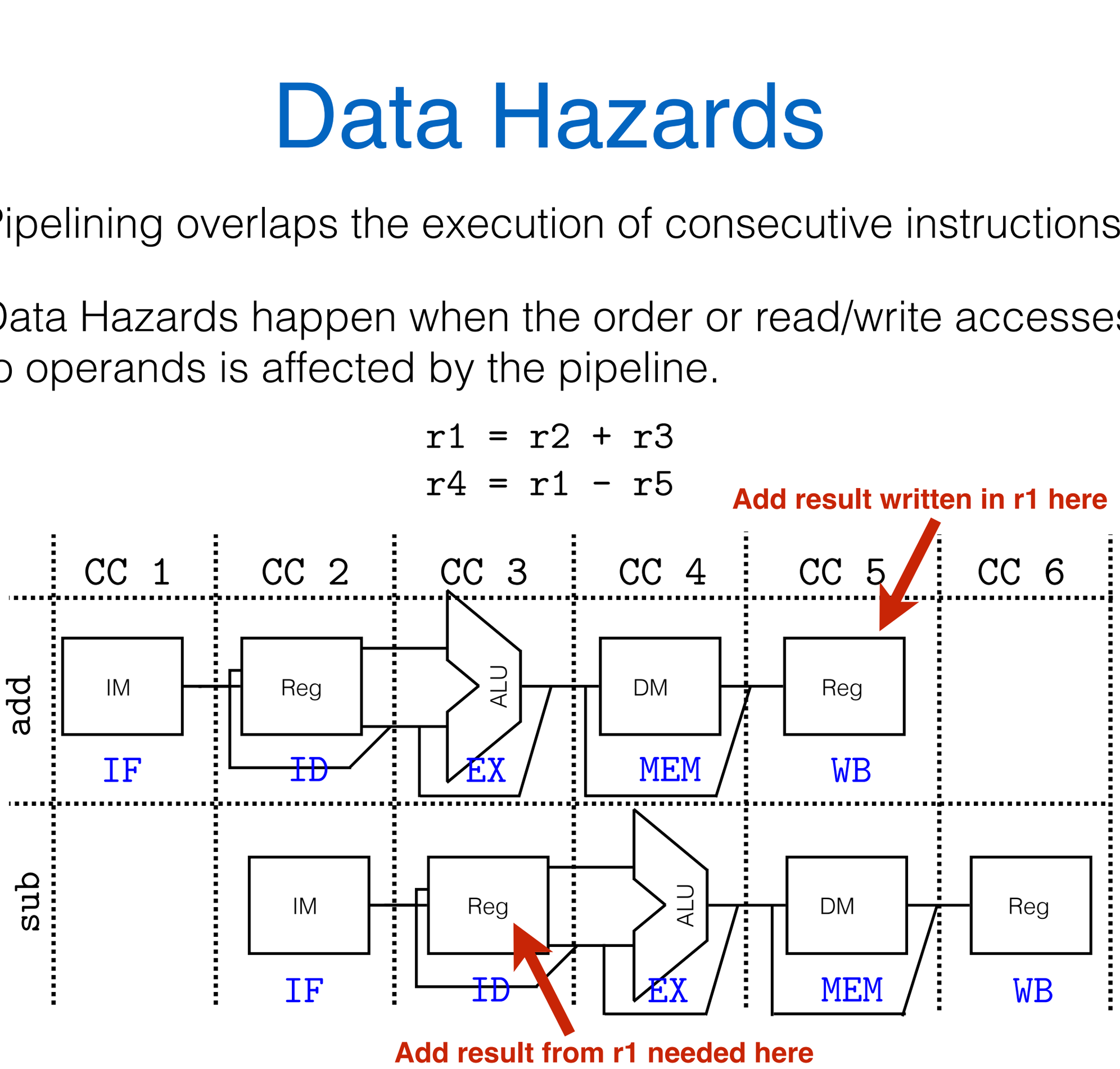
- Pipelining overlaps the execution of consecutive instructions.
- Data Hazards happen when the order or read/write accesses to operands is affected by the pipeline.

```
r1 = r2 + r3
r4 = r1 - r5
```

Data Hazards

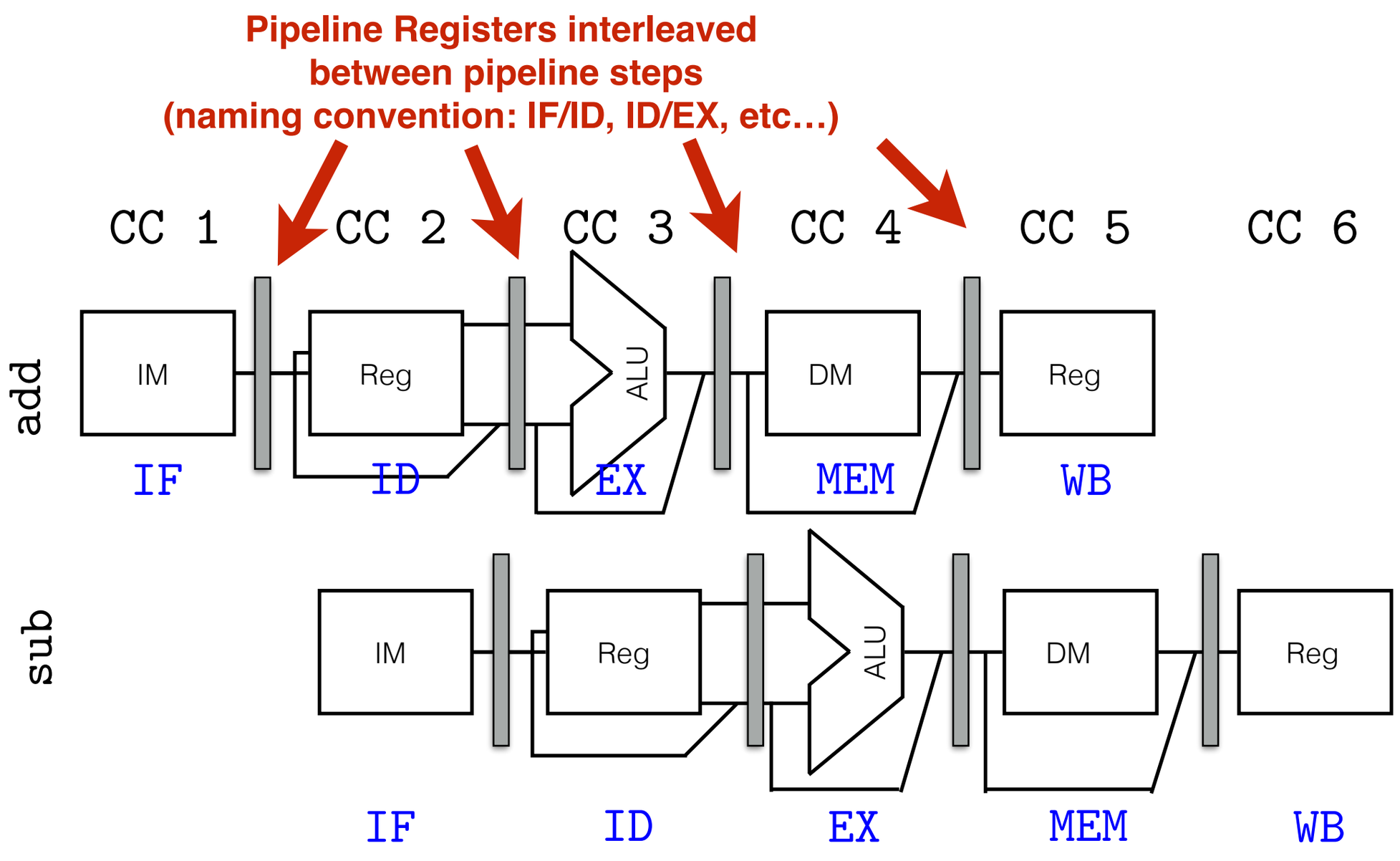
- Pipelining overlaps the execution of consecutive instructions.
- Data Hazards happen when the order or read/write accesses to operands is affected by the pipeline.

```
r1 = r2 + r3
r4 = r1 - r5
```



Data Hazards: Forwarding

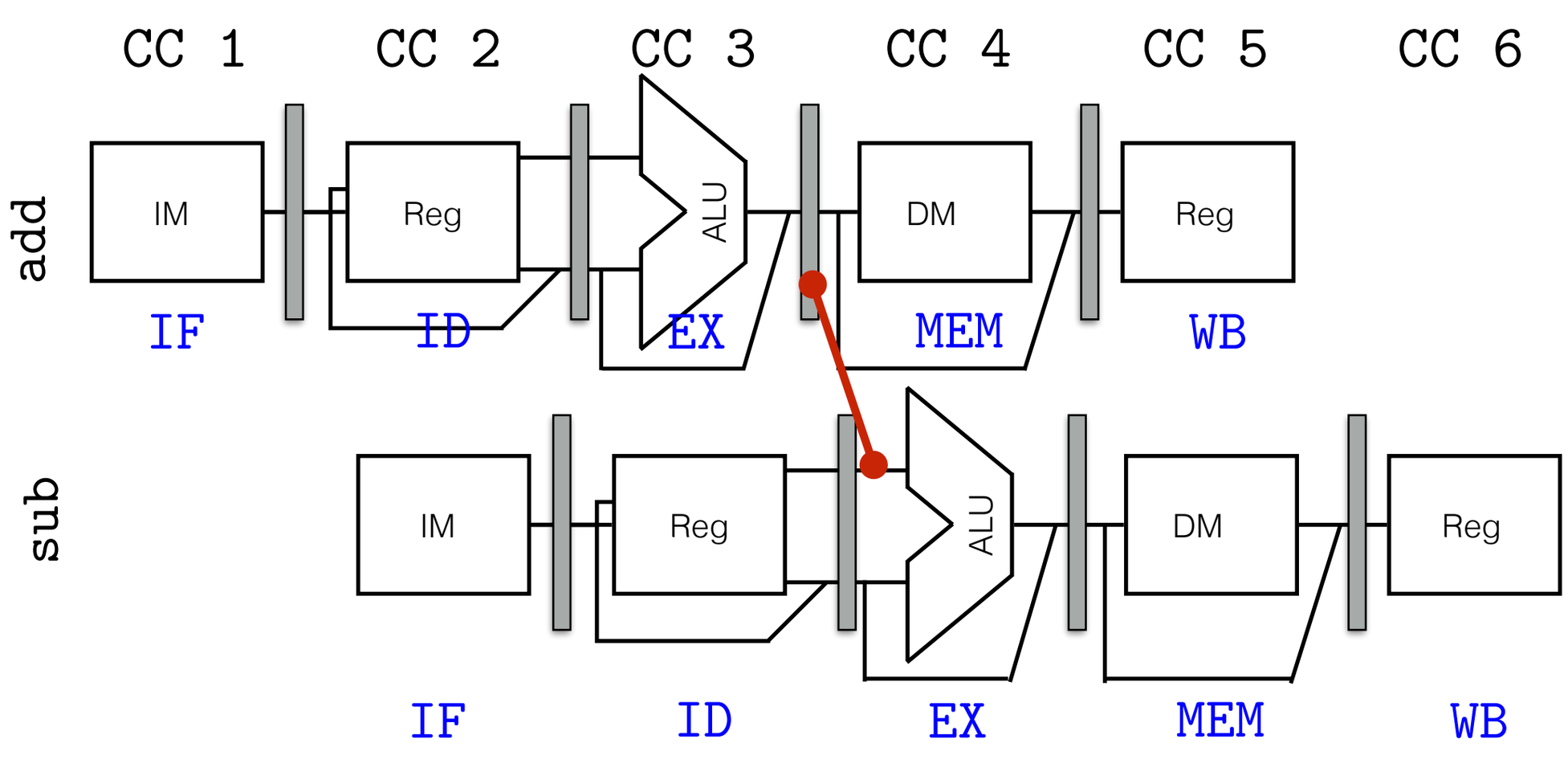
- Forwarding implemented to solve a family of data hazards.
- Key insight is that (in the previous example) the data has already be computed – just not yet stored in the actual register! Idea: forward this information to the ALU.



Data Hazards: Forwarding

- Forwarding implemented to solve a family of data hazards.
- Key insight is that (in the previous example) the data has already be computed – just not yet stored in the actual register! Idea: forward this information to the ALU.

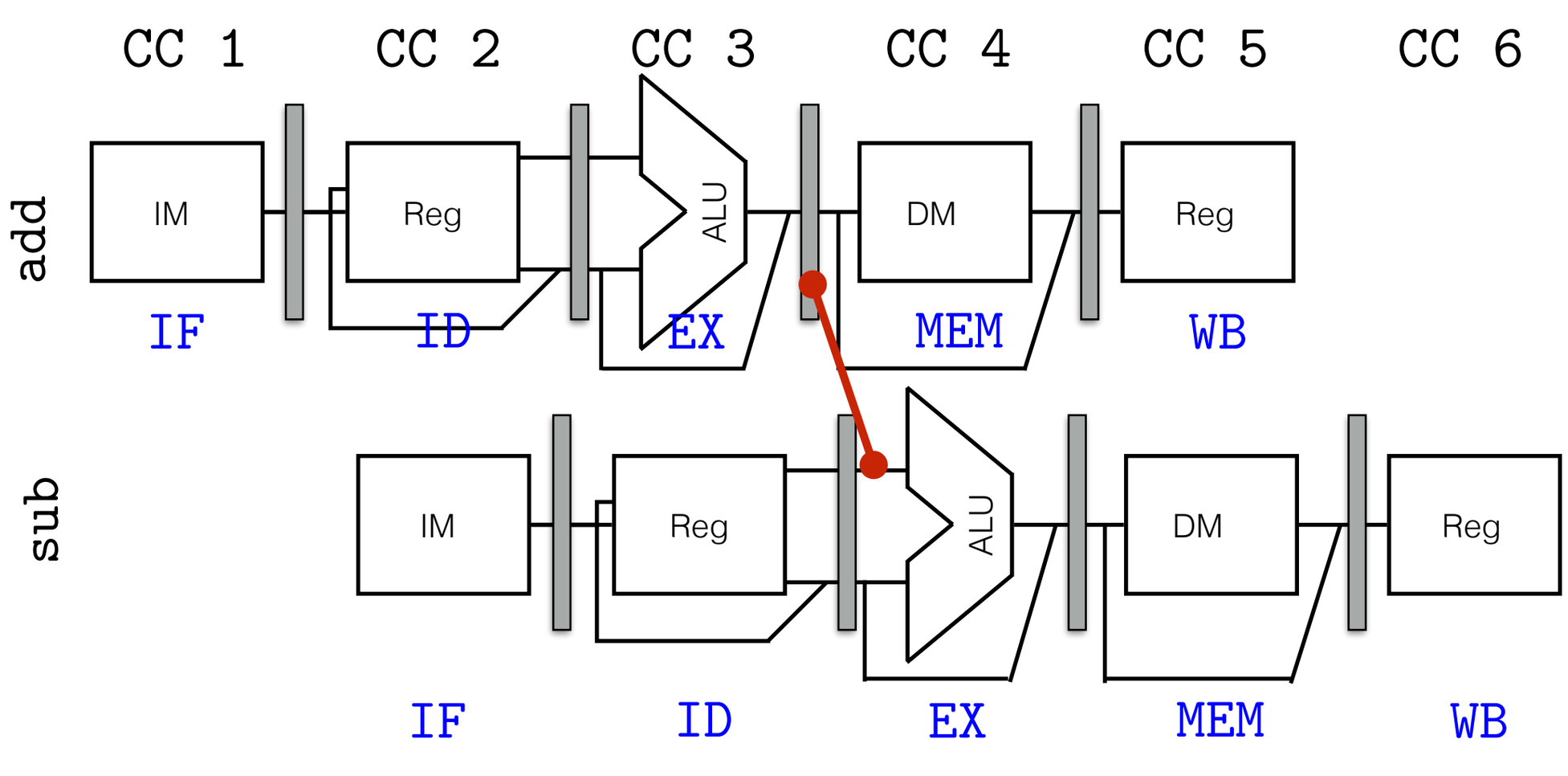
EX/MEM is forwarded to the ALU as possible input



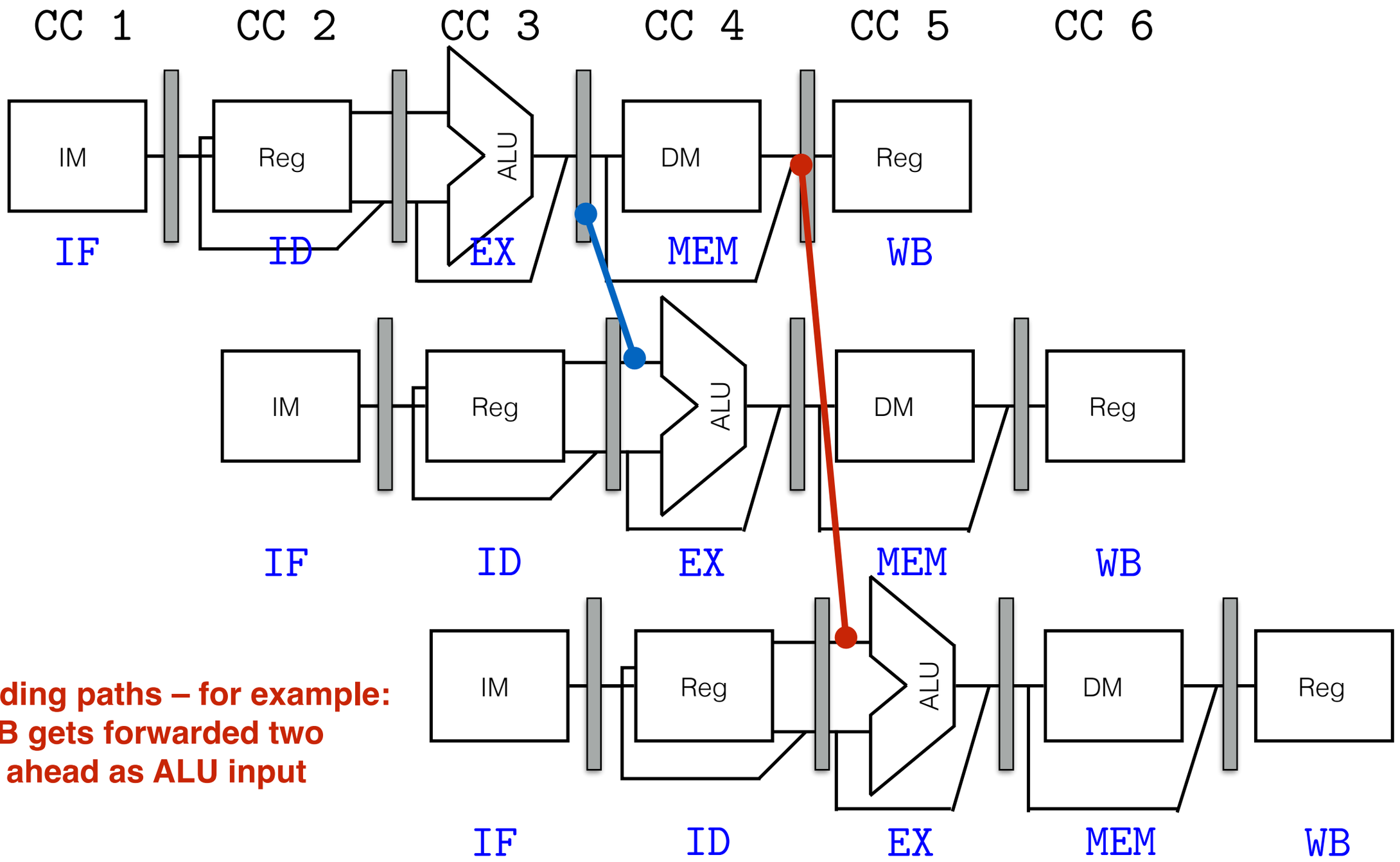
Data Hazards: Forwarding

- Forwarding implemented to solve a family of data hazards.
- Key insight is that (in the previous example) the data has already be computed – just not yet stored in the actual register! Idea: forward this information to the ALU.

EX/MEM is forwarded to the ALU as possible input



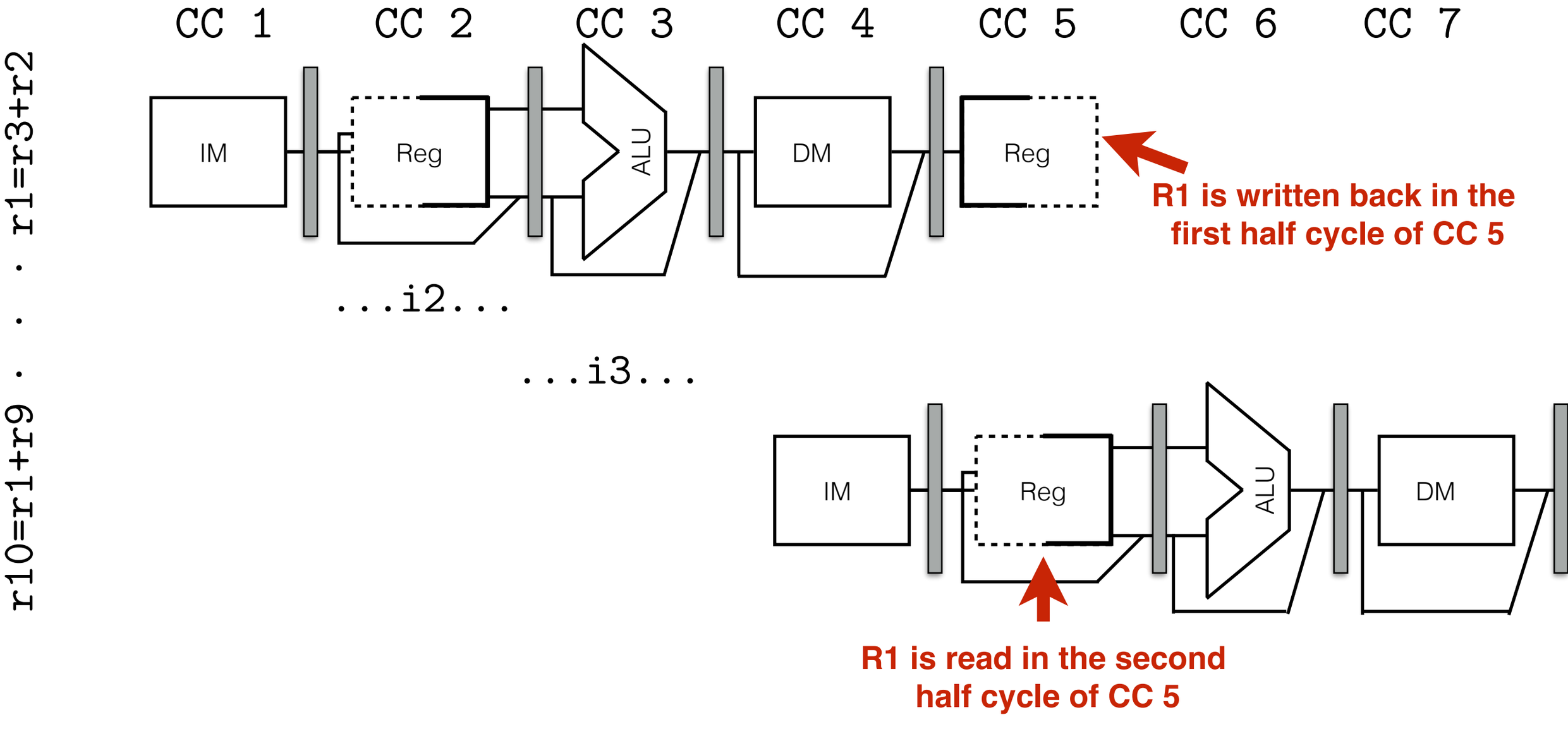
Data Hazards: Forwarding



Other forwarding paths – for example: MEM/WB gets forwarded two cycles ahead as ALU input

Data Hazards: Two-phase register access

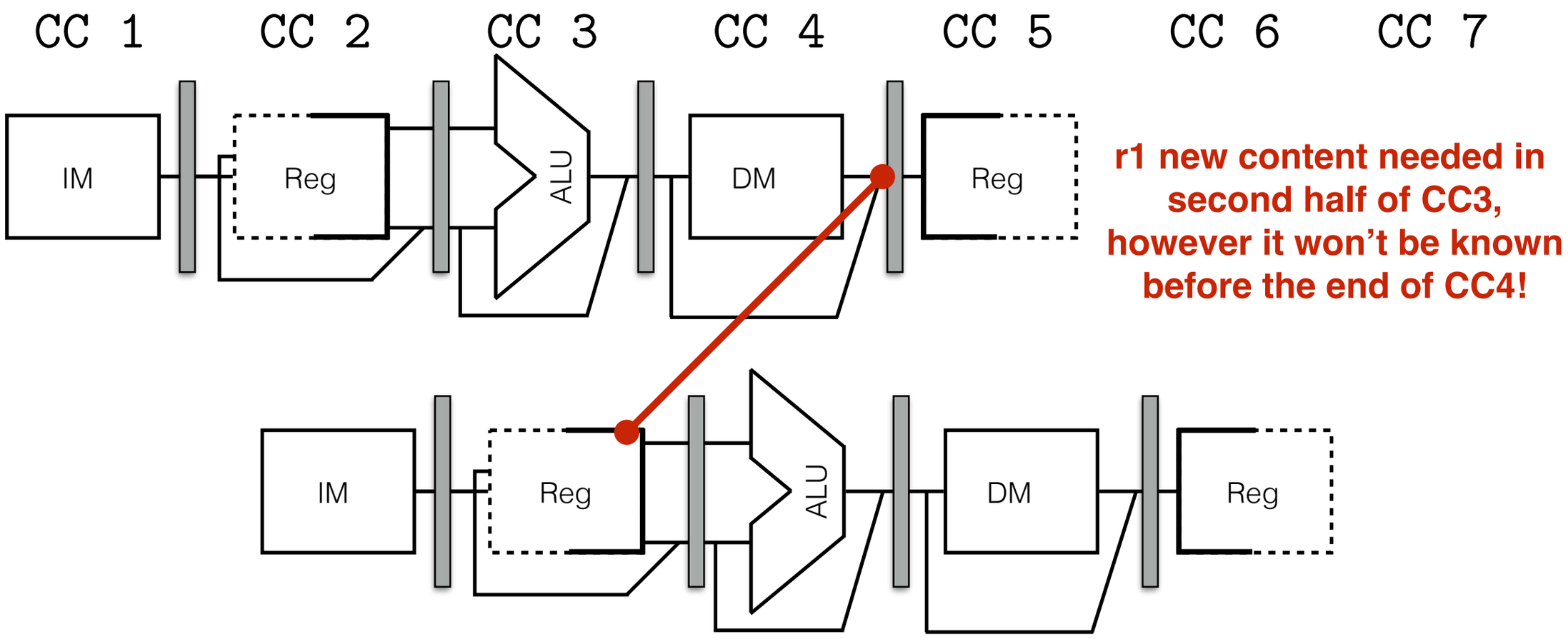
Another technique used for solving data hazards, is that of two-phase register access: Register **writing** take place in the first half of the clock cycle. Register **reading** in the second half.



Not All Data Hazards can be solved…

Forwarding and two-phase register access solves a lot of data hazards. But not all can be solved…

```
r1 = MEM_CONTENT[r2]
r3 = r4 + r1
```



r1 new content needed in second half of CC3, however it won't be known before the end of CC4!

Not All Data Hazards can be solved…

In these cases, nothing we can do but input a bubble in the pipeline

```
r1 = MEM_CONTENT[r2]
r3 = r4 + r1
```

CC 1	CC 2	CC 3	CC 4	CC 5	CC 6
IF	ID	EX	MEM	WB	
	IF	ID	stall	EX	MEM

MEM/WB is then forward as input to the ALU in step: EX.

Instruction Scheduling

- Sometimes data hazards cane be solved by the compiler scheduling instructions to avoid stalling [out-of-order execution]. Most modern compilers do this.

- Correctness must be ensured…

- NB: Software solution rather than hardware solution!

```
r1 = MEM_CONTENT[r2]
r3 = r4 + r1
r7 = r10 + r9
```

```
r1 = MEM_CONTENT[r2]
r7 = r10 + r9
r3 = r4 + r1
```

swap

CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7
IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB	
		IF	ID	EX	MEM	WB

No Stalling!

Control Hazards

- Control hazards arise when branch instruction need to be executed.
- If a branch is not taken the PC gets its normal update.
- If a branch is taken, then the PC needs to be updated by an offset.
- Of course it is in general impossible to know at compiling time what direction the branch is going to go.
- Branch hazards are a major hurdle in pipelining.

Branching in DLX/MIPS

- DLX doesn't rely on condition codes register. So normal registers are used to store the results that will decide the branching:

```
SLT   r1, r2, r3 //check r2<r3 and stores boolean results in r1
BEQZ  r1, L      //jump to L if r1==0
```

- Branch is resolved in the ID phase

```
IF    IR ← M[PC]; PC ← PC + 4
ID    if r_src == 0 (or r_src != 0) PC ← PC + offset
EX    idle
MEM   idle
WB    idle
```

Branching stall

- Branch doesn't get resolved until the end of second clock cycle. Leads to a stall.

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8
i=branch	IF	ID	EX	MEM	WB			
i+1		stall	IF	ID	EX	MEM	WB	
i+2				IF	ID	EX	MEM	WB
i+3					IF	ID	EX	MEM
i+4						IF	ID	EX

Waiting for branch resolution
in order to update PC

Can Something be done about branch stalls?

- Branches are a sizeable part of real-world programmes. Several strategies have been developed to ease the negative effect of branch stalls. We discuss:
  - Delayed Jumps [We have seen them in RISC-1].
  - Predict-not-taken [implemented in DLX/MIPS].
  - Dynamic branch prediction [we'll look at particular cases].

Predict-not-taken: The DLX branch pipeline

- It's a simple scheme: always predict that the branch is not taken and continue populating the pipeline accordingly.

- If it turns out that the branch is taken, we need to undo whatever had been done in the meanwhile, and re-fetch the correct instruction.

Correct prediction case (branch not taken)

```
i=branch      IF  ID  EX  MEM  WB
 i+1              IF  ID  EX  MEM  WB
 i+2                  IF  ID  EX  MEM  WB
```

Wrong prediction case (branch taken)

```
 i=branch      IF  ID  EX  MEM  WB
  i+1              IF idle idle idle idle
  i+2                  IF  ID  EX  MEM  WB
```

Comparing different branching strategies

- We have seen three strategies for mitigating control hazards (1 stall branches, branch not-taken, delayed branches). Which one performs best?
    - Assume: 14% Instruction are branchings; 65% of branches are taken; 50% chance of filling branch delay slot with useful operation.

Method	CC Stalls (average)	Clock cycles Per Instruction (CPI)
Stall Pipeline	1	`0.86 + 0.14 * 2 =` **`1.14`**
Predict-not-taken	0.65	`0.86 + 0.14(0.35 + 2*0.65) =` **`1.09`**
Delayed branch	0.5	`0.86 + 0.14*1.5 =` **`1.07`**

Dynamic Branch Prediction

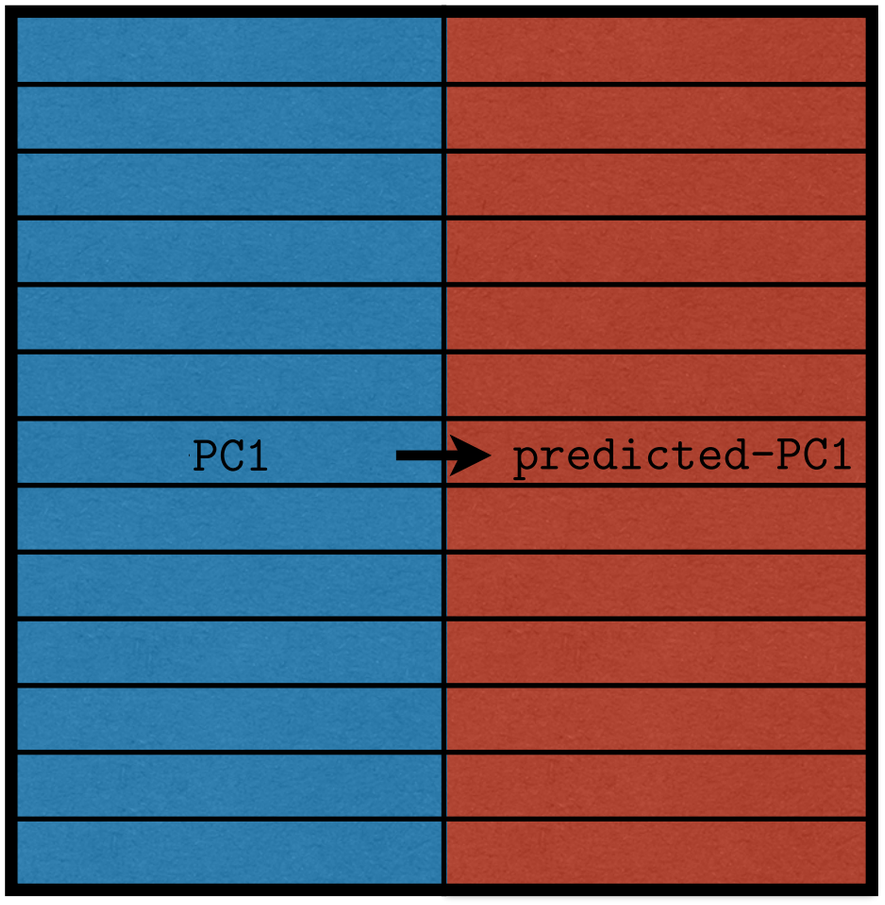
- So far we have seen simple scheme where it is either the compiler that partly takes care of the branch (delayed branch) or when the hardware defaults always to one choice (predict-not-taken).

- More sophisticated schemes take into account more information in order to predict what direction a branch will take, relying either on hardware or software implementation.

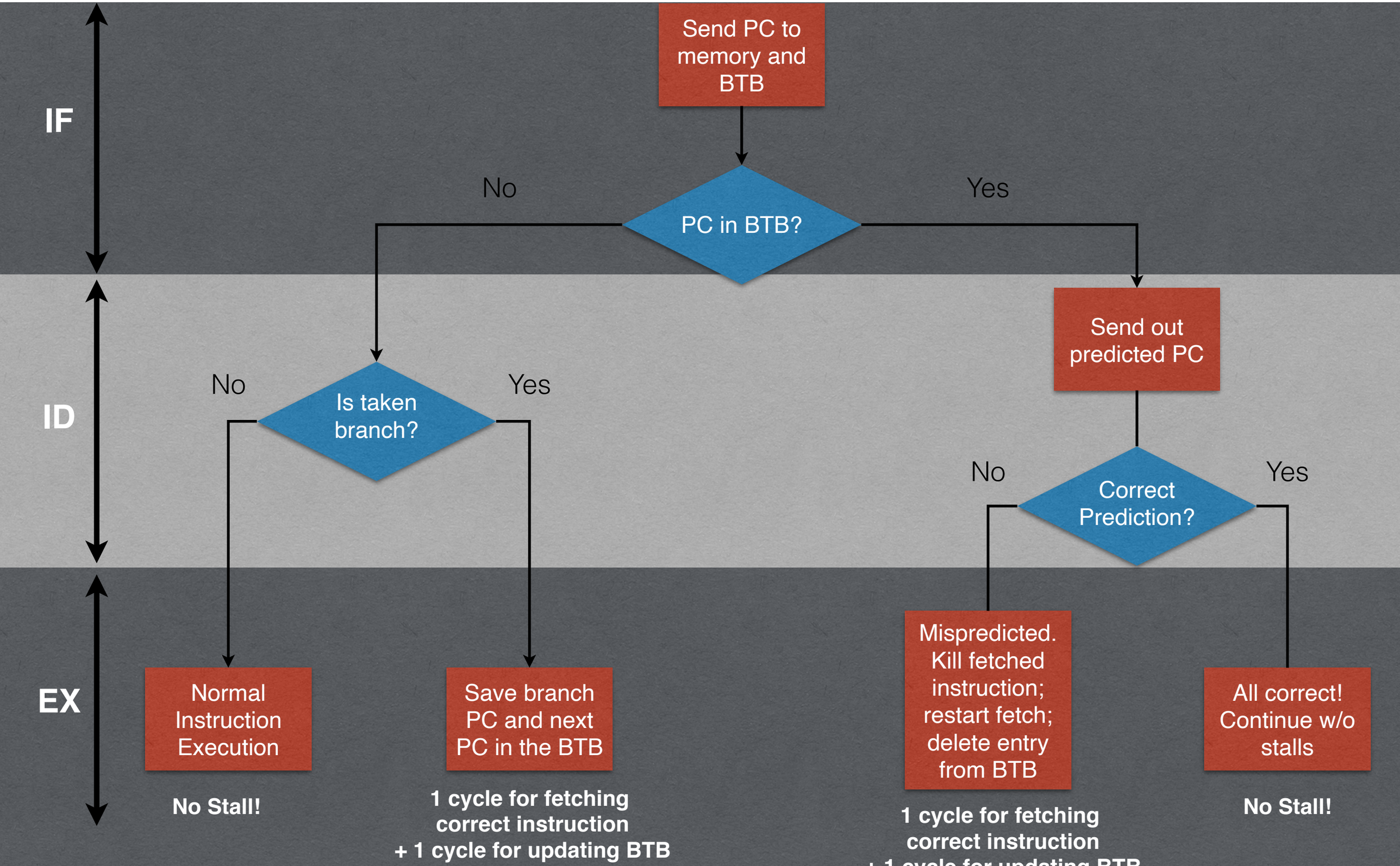
- We will briefly look at a simple hardware solution: branch-target buffers.

Branch-target buffers

- Try to predict if branch will be taken during IF phase.
- Implementation of a branch-target buffers: a look-up table of PC of branches already taken with corresponding resulting PC [almost like a cache for branches PC].
- If match found, then use the predicted PC for the next instruction.
- If branch is predicted correctly, then no stall!
- If branch predicted incorrectly, then must flush the pipeline and fetch the correct instruction.
- Update the branch-target buffer if branch prediction is wrong or if not in the buffer.



Notice that if the branch is not taken, then we simply need to go to the next instruction. No need to store it in the buffer then.



Two-bit branch prediction

- Notice that the branch-target buffer is updated every time the prediction is wrong… But what if the branch changes behaviour only for a single time?

```
L1:  add  r1, #10, r0
     ...
L2:  add  r1, r1 , #-1
     ...
     BNEZ r1, L2
     add  r2, r2, #-5
     BNEZ r2, L1
```

- The buffer gets updated twice in a row…
- It's a common settings: e.g., nested for loops.
- Idea: update buffer only if prediction is wrong two consecutive times.

Two-bit branch prediction

- Append two additional bit in each buffer entry to keep track of mispredictions.
- Update only if two consecutive misprediction are done.
- Can be generalised to more then two-bits (but performance results are similar)…