Computer organization and architecture

Lesson 18

Advanced microarchitecture

Advances in integrated circuit manufacturing have reduced transistor sizes.

Smaller transistors are faster and consume less power.

Even if the microarchitecture does not change, the clock frequency can increase because all the gates are faster.

Smaller transistors enable placing more transistors on a chip.

Power consumption increases with the number of transistors and the speed at which they operate.

Power consumption is now an essential concern.

Microprocessor designers have a challenging task juggling the trade-offs among speed, power, and cost for chips with billions of transistors.

Deep pipelines

10–20 stages are now commonly used.

The maximum number of pipeline stages is limited by

– **Pipeline hazards**: longer pipelines introduce more dependencies.

Some of the dependencies can be solved by forwarding but others require stalls, which increase the CPI.

– **Sequencing overhead**: the pipeline registers between each stage have sequencing overhead from their setup time and clk-to-Q delay (as well as clock skew).

– **cost**: Extra pipeline registers and hardware required to handle hazards.

Example: calculate the optimal number of pipeline stages

Consider building a pipelined processor by chopping up the single-cycle processor into *N* stages.

The single-cycle processor has a propagation delay of 740 ps through the combinational logic.

The sequencing overhead of a register is 90 ps.

Assume that the combinational delay can be arbitrarily divided into any number of stages and that pipeline hazard logic does not increase the delay.

Assume that the five-stage pipeline has a CPI of 1.23.

Assume that each additional stage increases the CPI by 0.1 because of branch mispredictions and other pipeline hazards.

How many pipeline stages should be used to make the processor execute programs as fast as possible?

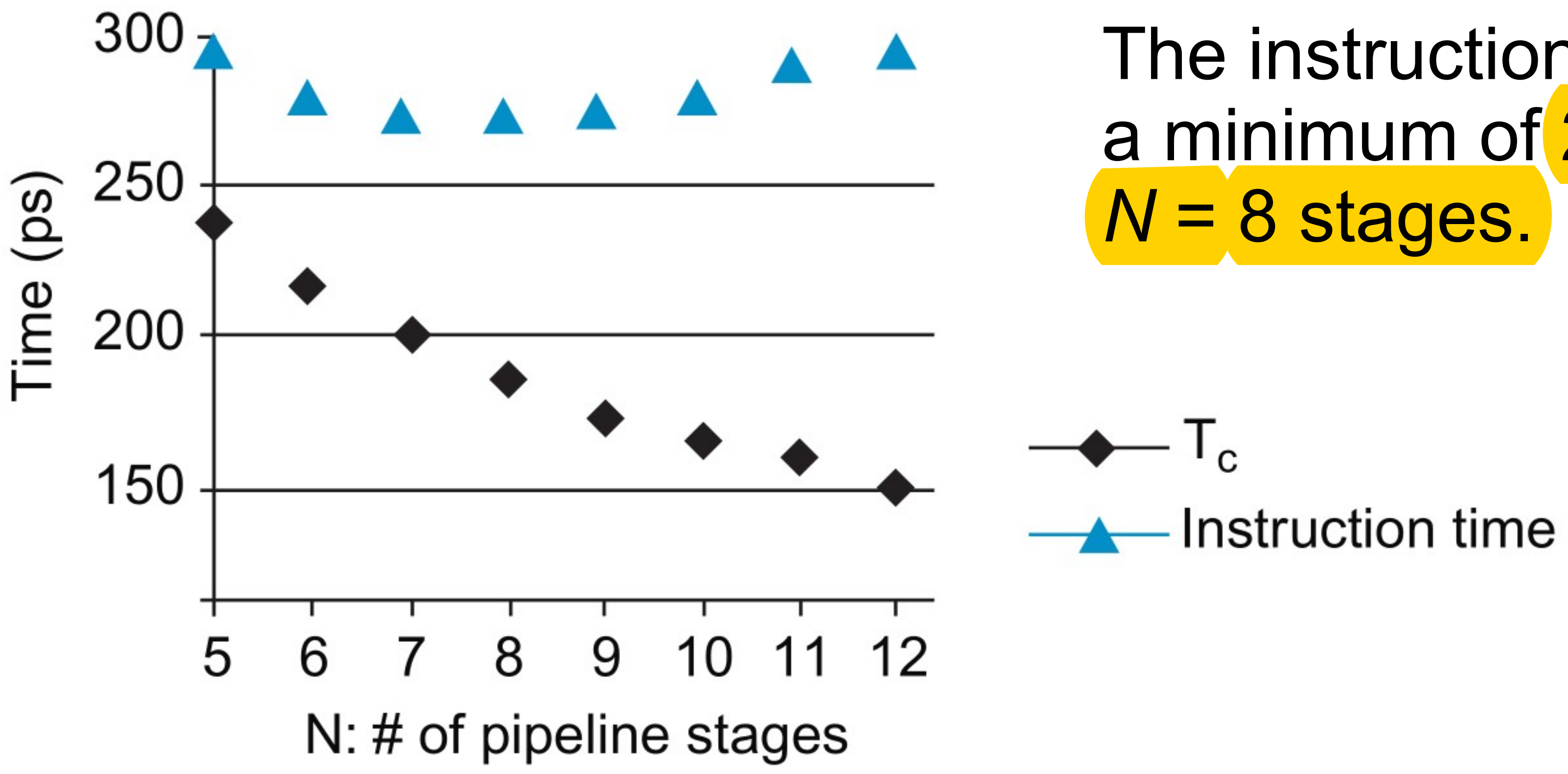
Solution: The cycle time for an *N*-stage pipeline is

$$T_c = (740/N + 90)\,ps$$

The CPI is $CPI = 1.23 + 0.1(N-5)$

The time per instruction is

$$T_I = T_c CPI = (740/N + 90)(1.23 + 0.1(N-5))$$



The instruction time has a minimum of 279 ps at *N* = 8 stages.

Micro-operations

Pure RISC architectures such as MIPS contain only simple instructions, typically those that can be executed in a single cycle on a simple, fast datapath with a three-ported register file, single ALU, and single data memory access.

CISC architectures include instructions requiring more registers, more additions, or more than one memory access per instruction.

The x86 instruction `ADD [ESP], [EDX + 80 + EDI*2]`

involves reading the 3 registers, adding the base, displacement, and scaled index, reading 2 memory locations, summing their values, and writing the result back to memory.

A microprocessor that could perform all of these functions at once would be unnecessarily slow on more common, simpler instructions.

Computer architects make the common case fast by defining a set of simple **micro-operations** (also known as **micro-ops** or **μops**) that can be executed on simple datapaths.

Each real instruction is decoded into one or more μops.

For example, if we defined µops resembling basic ARM instructions and some temporary registers T1 and T2 for holding intermediate results, then the x86 instruction

```
        ADD [ESP], [EDX + 80 + EDI*2]
```

could become seven µops:

```
ADD T1, [EDX + 80] ; T1 <- EDX + 80
LSL T2, EDI, 2 ; T2 <- EDI*2
ADD T1, T2, T2 ; T1 <- EDX + 80 + EDI*2
LDR T1, [T1]   ; T1 <- MEM[EDX + 80 + EDI*2]
LDR T2, [ESP]  ; T2 <- MEM[ESP]
ADD T1, T2, T1 ; T1 <- MEM[ESP] + MEM[EDX + 80 + EDI*2]
STR T1, [ESP]  ; MEM[ESP]<- MEM[ESP] + MEM[EDX + 80 + EDI*2]
```

Although most ARM instructions are simple, some are decomposed into multiple micro-ops as well.

For example, loads with postindexed addressing (such as `LDR R1, [R2], #4`) require a second write port on the register file.

Data-processing instructions with register-shifted register addressing (such as `ORR R3, R4, R5, LSL R6`) require a third read port on the register file.

Instead of providing a larger five-port register file, the ARM datapath may decode these complex instructions into pairs of simpler instructions.

Complex Op	Micro-op Sequence
`LDR R1, [R2], #4`	`LDR R1, [R2]`
	`ADD R2, R2, #4`
`ORR R3, R4, R5 LSL R6`	`LSL T1, R5, R6`
	`ORR R3, R4, T1`

Although the programmer could have written the simpler instructions directly and the program may have run just as fast, a single complex instruction takes less memory than the pair of simpler instructions.

Reading instructions from external memory can consume significant power, so the complex instruction also can save power.

Microarchitects make the decision of whether to provide hardware to implement a complex operation directly or break it into micro-op sequences.

These choices lead to different points in the performance-power-cost design space.

The ARM instruction set is so successful in part because of the architects’ judicious choice of instructions that give better code density than pure RISC instructions sets such as MIPS, yet more efficient decoding than CISC instruction sets such as x86.

Branch prediction

The branch misprediction penalty is a major reason for increased CPI.

As pipelines get deeper, branches are resolved later in the pipeline.

The branch misprediction penalty gets larger because all the instructions issued after the mispredicted branch must be flushed.

To address this problem, most pipelined processors use a branch predictor to guess whether the branch should be taken.

Recall that the pipeline microprocessor from the lesson 16 simply predicted that branches are never taken.

Some branches occur when a program reaches the end of a loop and branches back to repeat the loop (e.g., in a for or while loop).

Loops tend to be executed many times, so these backward branches are usually taken.

The simplest form of branch prediction checks the direction of the branch and predicts that backward branches should be taken.

This is called **static branch prediction**, because it does not depend on the history of the program.

Forward branches are difficult to predict without knowing more about the specific program.

Most processors use **dynamic branch predictors**, which use the history of program execution to guess whether a branch should be taken.

Dynamic branch predictors maintain a table of the last several hundred (or thousand) branch instructions that the processor has executed.

The table, called a **branch target buffer**, includes the destination of the branch and a history of whether the branch was taken.

Example

A **one-bit dynamic branch predictor** remembers whether the branch was taken the last time and predicts that it will do the same thing the next time.

While the loop is repeating, it remembers that the `BGE` was not taken last time and predicts that it should not be taken next time.

```
        MOV R1, #0
        MOV R0, #0
FOR
        CMP R0, #10
        BGE DONE
        ADD R1, R1, R0
        ADD R0, R0, #1
        B FOR
DONE
```

This is a correct prediction until the last branch of the loop, when the branch does get taken.

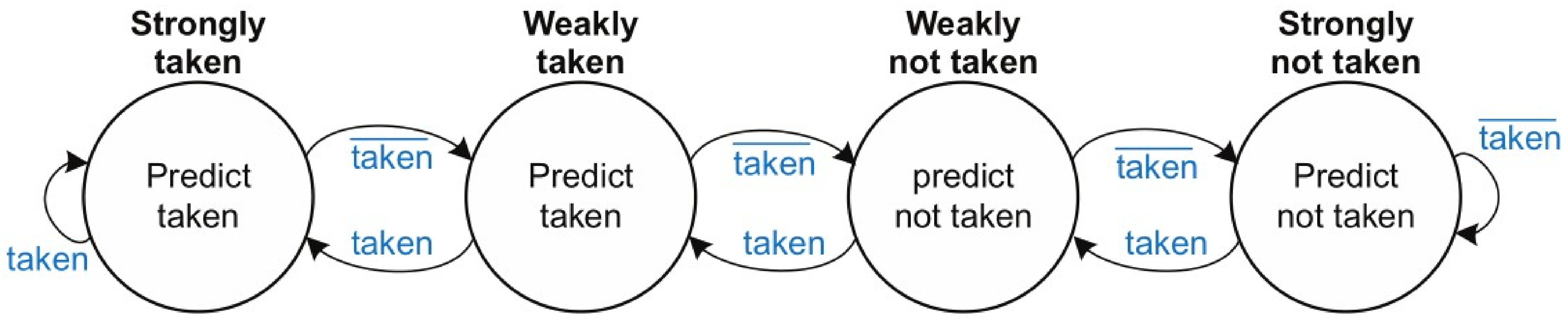
If the loop is run again, the branch predictor remembers that the last branch was taken.

Therefore, it incorrectly predicts that the branch should be taken when the loop is first run again.

A 1-bit branch predictor mispredicts the first and last branches of a loop.

```
     MOV R1, #0
     MOV R0, #0
FOR
     CMP R0, #10
     BGE DONE
     ADD R1, R1, R0
     ADD R0, R0, #1
     B FOR
DONE
```

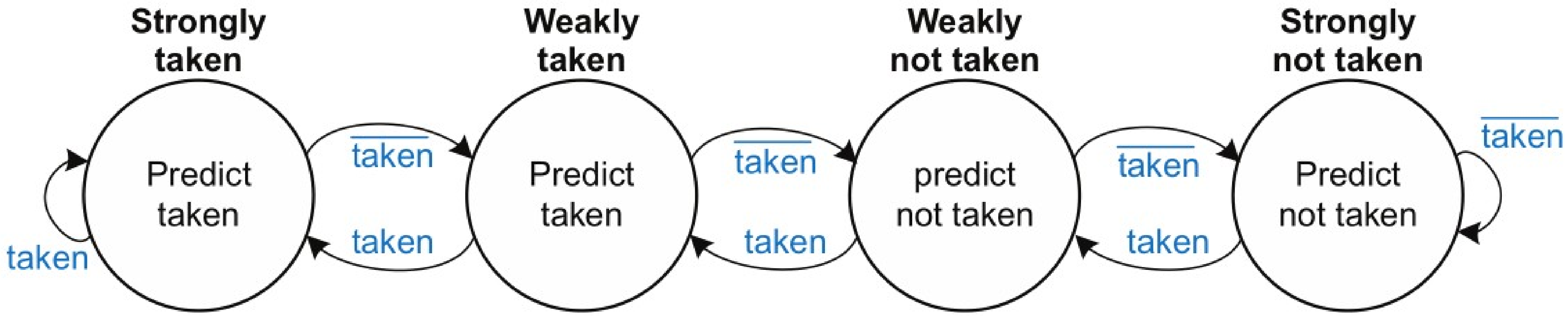
A **two-bit dynamic branch predictor** solves this problem by having four states:



When the loop is repeating, it enters the “strongly not taken” state and predicts that the branch should not be taken next time.

This is correct until the last branch of the loop, which is taken and moves the predictor to the “weakly not taken” state.

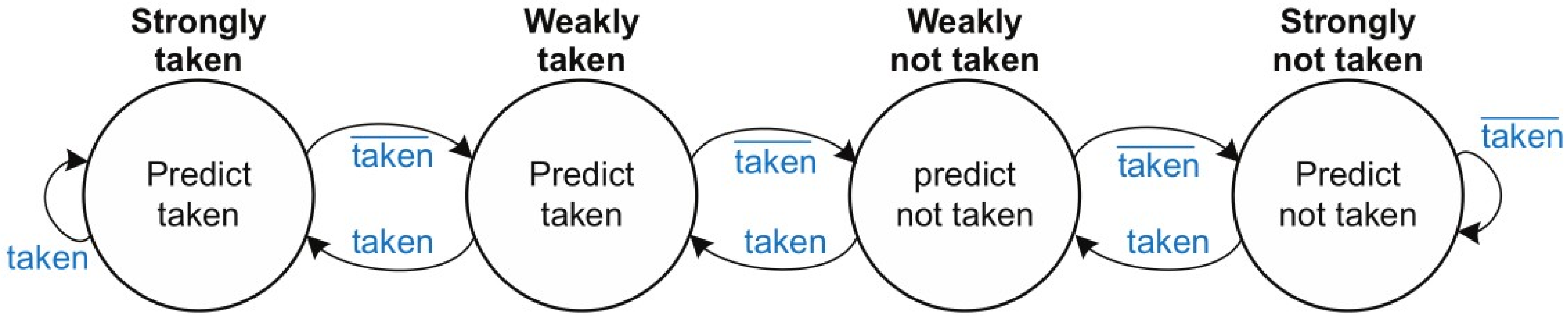
```
          MOV R1, #0
          MOV R0, #0
FOR
          CMP R0, #10
          BGE DONE
          ADD R1, R1, R0
          ADD R0, R0, #1
          B FOR
DONE
```



When the loop is first run again, the branch predictor correctly predicts that the branch should not be taken and re-enters the “strongly not taken” state.

```
     MOV R1, #0
     MOV R0, #0
FOR
     CMP R0, #10
     BGE DONE
     ADD R1, R1, R0
     ADD R0, R0, #1
     B FOR
DONE
```

A two-bit branch predictor mispredicts only the last branch of a loop.



The branch predictor operates in the Fetch stage of the pipeline so that it can determine which instruction to execute on the next cycle.

When it predicts that the branch should be taken, the processor fetches the next instruction from the branch destination stored in the branch target buffer.

Branch predictors may be used to track even more history of the program to increase the accuracy of predictions.

Good branch predictors achieve better than 90% accuracy on typical programs.

Scalar, vector, superscalar

A **scalar processor** acts on one piece of data at a time.

A **vector processor** acts on several pieces of data with a single instruction.

A **superscalar processor** issues several instructions at a time, each of which operates on one piece of data.

Vector processors are heavily used in graphics processing units (GPUs).

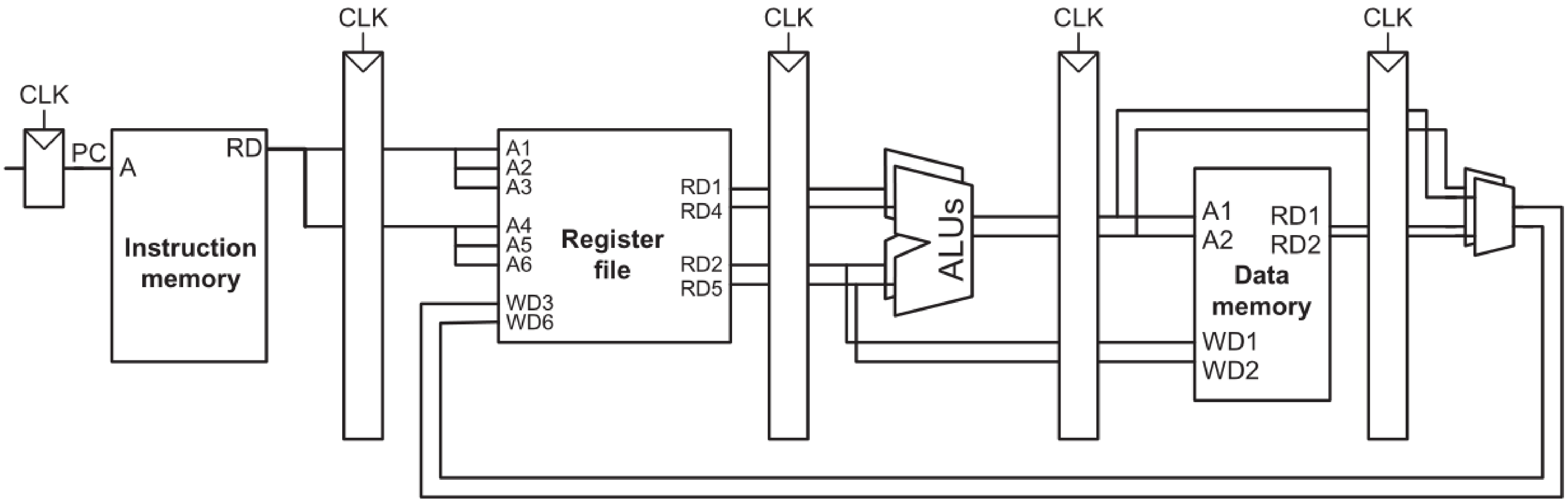
Modern high-performance microprocessors are superscalar, because issuing several independent instructions is more flexible than processing vectors.

Superscalar processor

A **superscalar processor** contains multiple copies of the datapath hardware to execute multiple instructions simultaneously.

Multiple execution units is a case of **spatial parallelism**.

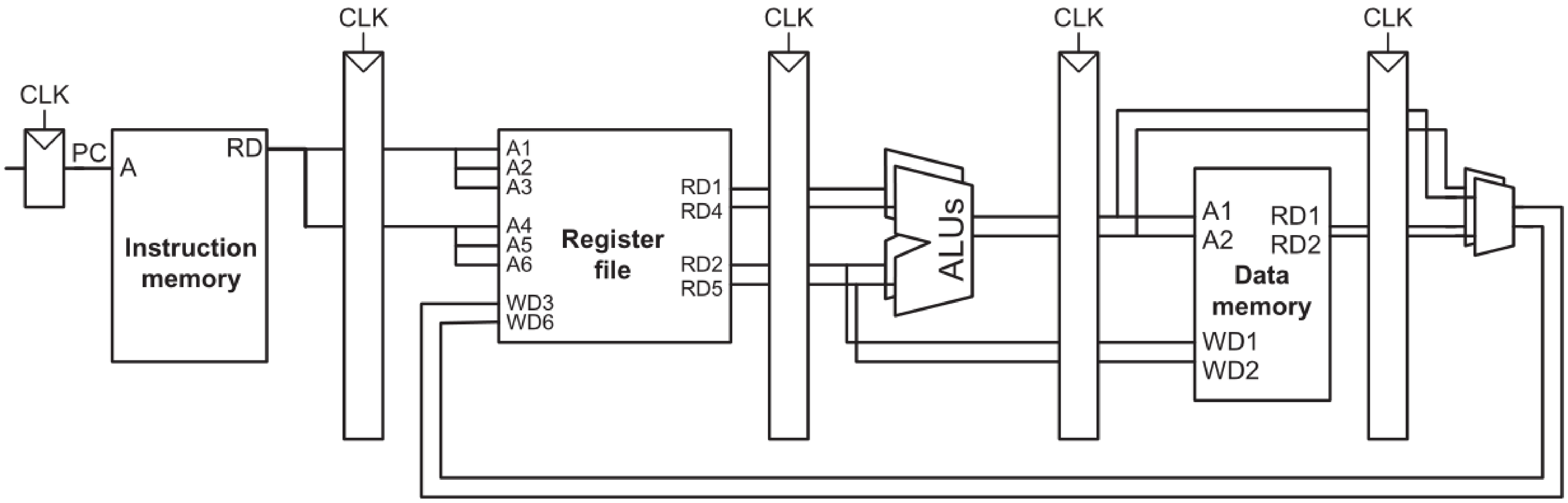
A superscalar processor that fetches and executes two instructions per cycle:



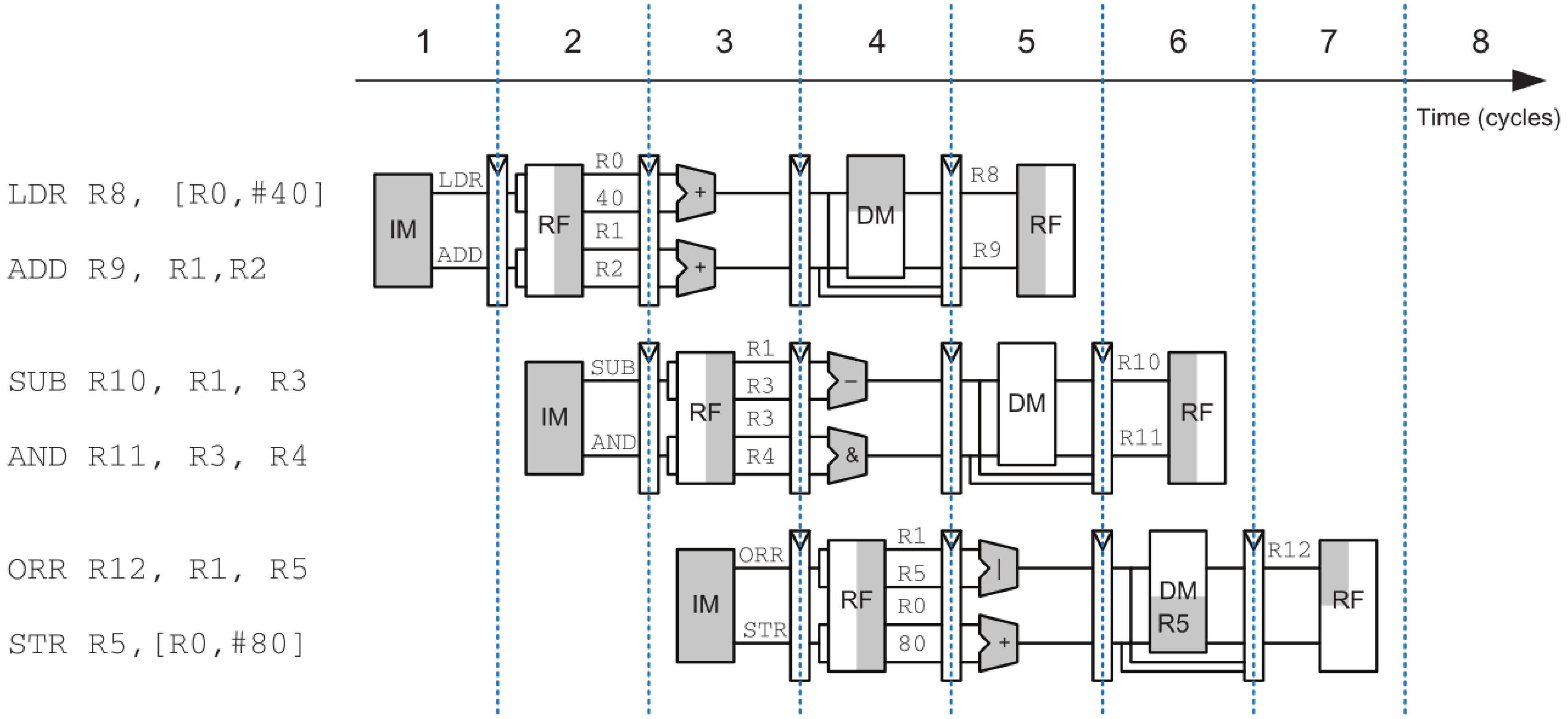
The datapath fetches two instructions at a time from the instruction memory.

It has a six-ported register file to read four source operands and write two results back in each cycle.

It also contains two ALUs and a two-ported data memory to execute the two instructions at the same time.



A pipeline diagram of the super-scalar processor executing two instructions on each cycle.



Both **temporal** and **spatial** forms of parallelism are here.

For this program, the processor has a CPI of 0.5.

The reciprocal of the CPI is IPC – the instructions per cycle.

This processor has an IPC of 2 on this program.

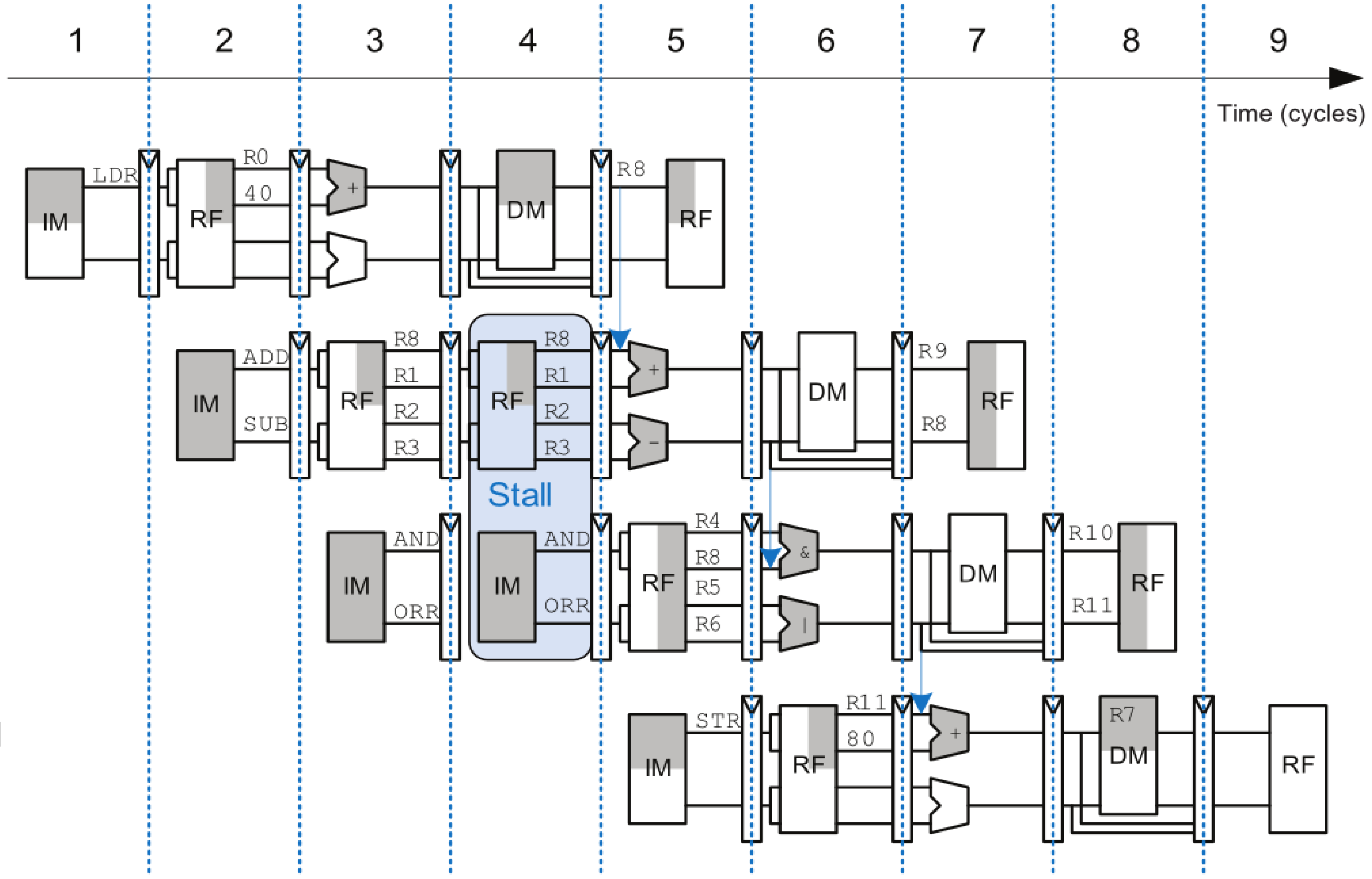
Executing many instructions simultaneously is difficult because of dependencies.

```
LDR R8, [R0, #40]

ADD R9, R8, R1
SUB R8, R2, R3

AND R10, R4, R8
ORR R11, R5, R6

STR R7, [R11, #80]
```



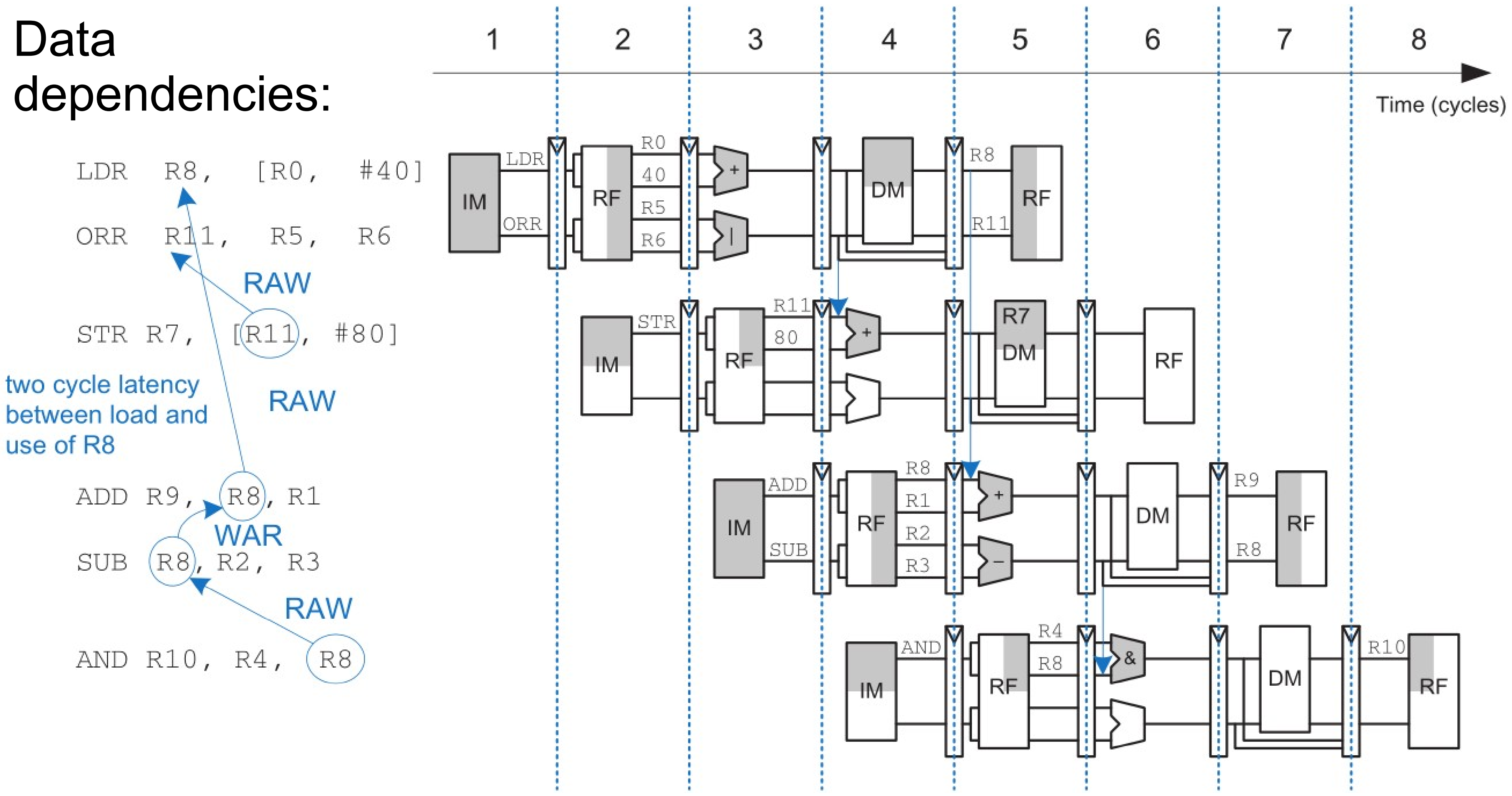
This program requires five cycles to issue six instructions, for an IPC of 1.2

Out-of-order processor

To cope with the problem of dependencies, an **out-of-order processor** looks ahead across many instructions to issue, or begin executing, independent instructions as rapidly as possible.

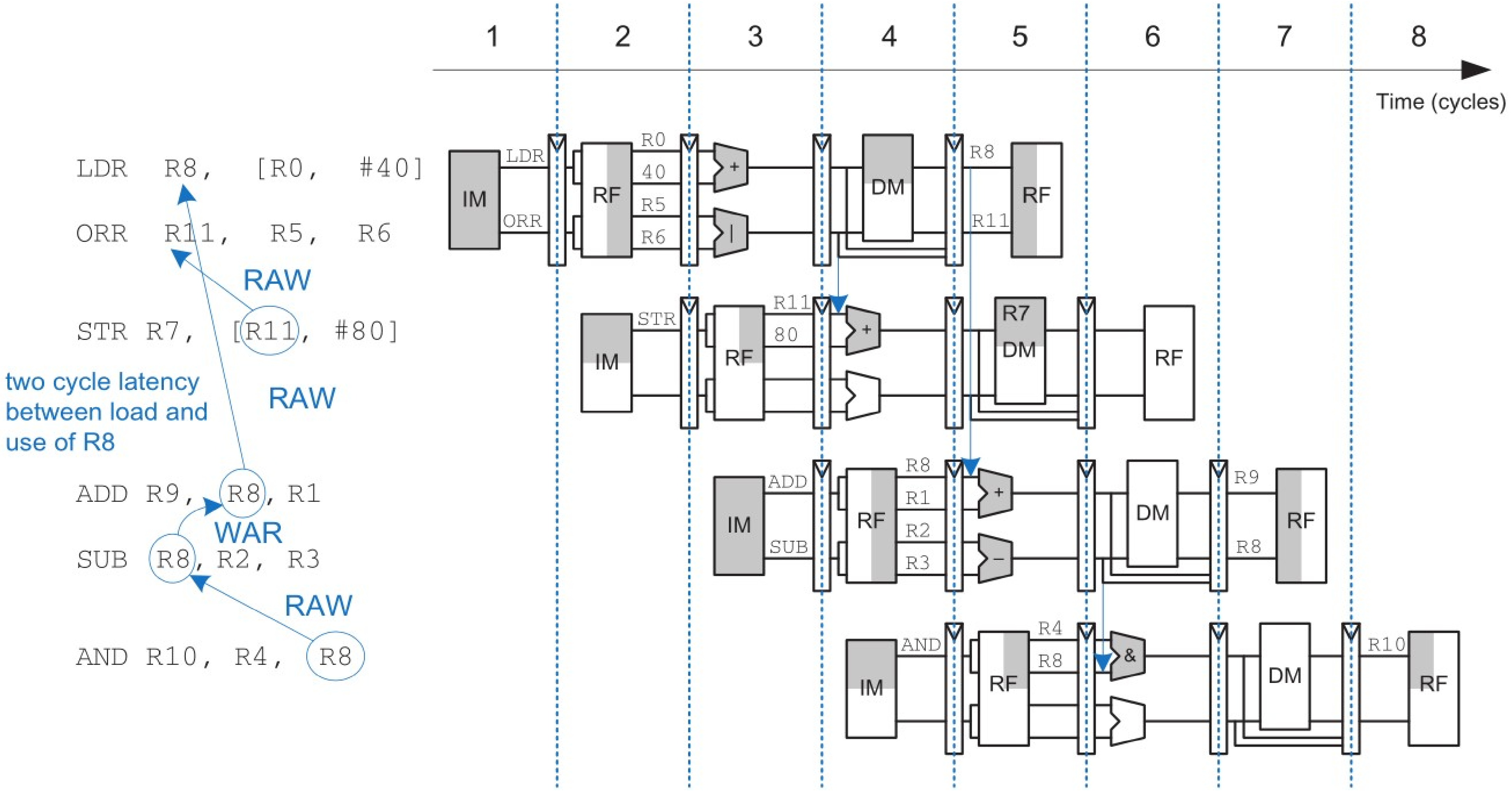
The instructions can issue in a different order than that written by the programmer, as long as dependencies are honored so that the program produces the intended result.

Data dependencies:



The dependence of ADD on LDR by way of R8 is a **read after write** (**RAW**) hazard.

ADD must not read R8 until after LDR has written it.



The dependence between SUB and ADD by way of R8 is called a **write after read** (**WAR**) hazard or an **antidependence**.

SUB must not write R8 before ADD reads R8

WAR hazards could not occur in the simple pipeline, but may happen in an out-of-order processor if the dependent instruction (in this case, `SUB`) is moved too early.

A third type of hazard: a **write after write** (**WAW**) hazard or an **output dependence**.

A WAW hazard occurs if an instruction attempts to write a register after a subsequent instruction has already written it.

The hazard would result in the wrong value being written to the register.

```
LDR R8, [R3]
ADD R8, R1, R2
```

The final value in R8 should come from `ADD` according to the order of the program.

If an out-of-order processor attempted to execute `ADD` first, then a WAW hazard would occur.

Out-of-order processors use a table to keep track of instructions waiting to issue.

The table, called a **scoreboard**, contains information about the dependencies.

The size of the table determines how many instructions can be considered for issue.

On each cycle, the processor examines the table and issues as many instructions as it can, limited by the dependencies and by the number of execution units (e.g., ALUs, memory ports) that are available.

The **instruction level parallelism** (**ILP**) is the number of instructions that can be executed simultaneously for a particular program and microarchitecture.

Register renaming

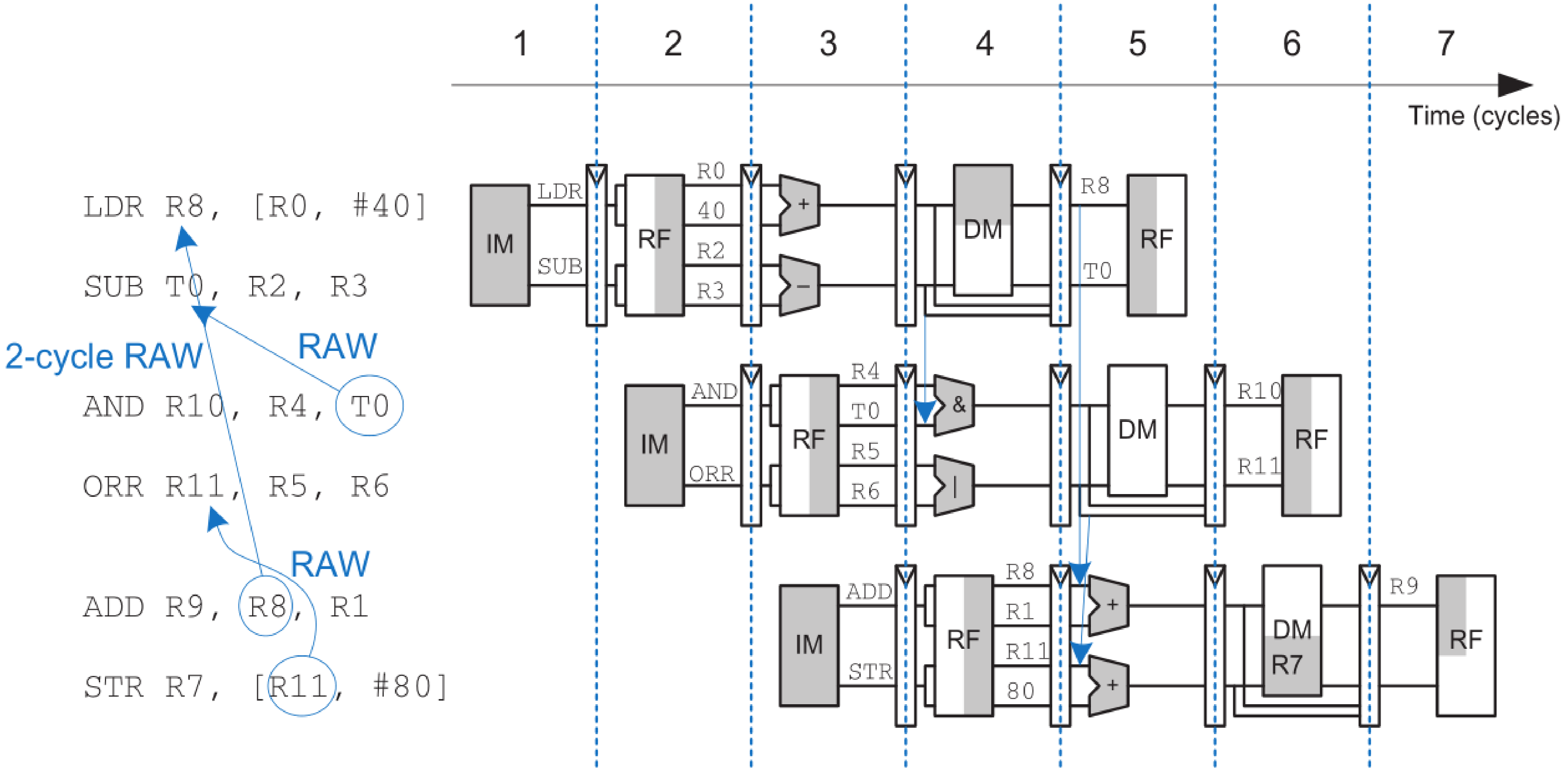
Out-of-order processors use a technique called **register renaming** to eliminate WAR and WAW hazards.

Register renaming adds some nonarchitectural renaming registers to the processor.

For example, a processor might add 20 renaming registers, called T0–T19.

The programmer cannot use these registers directly, because they are not part of the architecture.

However, the processor is free to use them to eliminate hazards.



The out-of-order processor could rename R8 to T0 for the SUB instruction.

Then, SUB could be executed sooner, because T0 has no dependency on the ADD instruction

6 instructions in 3 cycles: IPC = 2

Multithreading

A multithreaded processor contains more than one copy of its architectural state, so that more than one thread can be active at a time.

For example, if we extended a processor to have four program counters and 64 registers, four threads could be available at one time.

If one thread stalls while waiting for data from main memory, then the processor could **context switch** to another thread without any delay, because the program counter and registers are already available.

If one thread lacks sufficient parallelism to keep all the execution units busy in a superscalar design, then another thread could issue instructions to the idle units.

Multiprocessors

A multiprocessor system consists of multiple processors and a method for communication between the processors.

Three common classes of multiprocessors:

- **symmetric** (or **homogeneous**) multiprocessors
- **heterogeneous** multiprocessors
- **clusters**

Symmetric multiprocessors

Symmetric multiprocessors include two or more identical processors sharing a single main memory.

The multiple processors may be separate chips or multiple cores on the same chip.

Advantages:

– simple to design because the processor can be designed once and then replicated multiple times

– programming for and executing code on a symmetric multiprocessor is straightforward because any program can run on any processor in the system and achieve approximately the same performance.

Heterogeneous multiprocessors

Continuing to add more and more symmetric cores is not guaranteed to provide continued performance improvement.

Heterogeneous multiprocessors incorporate different types of cores and/or specialized hardware in a single system.

Each application uses those resources that provide the best performance, or power-performance ratio, for that application.

A heterogeneous system can incorporate cores with different microarchitectures that have different power, performance, and area trade-offs.

big.LITTLE

– a heterogeneous strategy popularized by ARM

A system contains both energy-efficient and high-performance cores.

“LITTLE” cores such as the Cortex-A53 are single-issue or dual-issue in-order processors with good energy efficiency that handle routine tasks.

“big” cores such as the Cortex-A57 are more complex superscalar out-of-order cores delivering high performance for peak loads.

Accelerators

Another heterogeneous strategy in which a system contains special-purpose hardware optimized for performance or energy efficiency on specific types of tasks.

Example: a mobile system-on-chip (SoC) may contain dedicated accelerators for graphics processing, video, wireless communication, real-time tasks, and cryptography.

These accelerators can be 10–100x more efficient than general-purpose processors for the same tasks.

Digital signal processors are another class of accelerators that have a specialized instruction set optimized for math-intensive tasks.

ARM DynamIQ

http://pages.arm.com/dynamiq-technology.html

An evolutionary step forward for ARM big.LITTLE technology

– multi-core, each core with different performance and power characteristics

– new dedicated processor instructions for Machine Learning and Artificial Intelligence

Drawbacks of heterogeneous systems

– difficult to design the different heterogeneous elements

– more programming effort to decide when and how
to make use of the varying resources

Clusters

In **clustered multiprocessors**, each processor has its own local memory system.

One type of cluster is a group of personal computers connected together on the network running software to jointly solve a large problem.

Another type of cluster that has become very important is the data center, in which racks of computers and disks are networked together and share power and cooling.