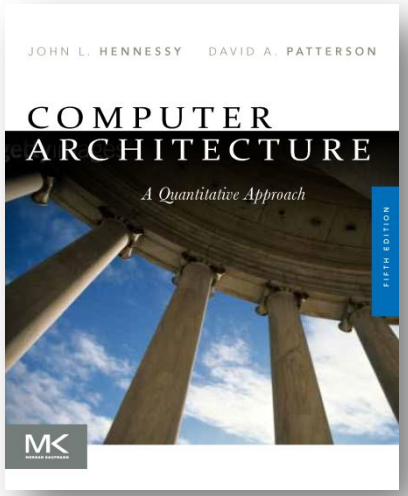
PADP(18CS73)
Unit 1

Dr. Minal Moharir



Chapter 1

Fundamentals of Quantitative Design and Analysis

Computer Technology Today

Computer Technology today has made incredible progress in roughly Sixty Five Years after the first general purpose Electronic Computer was created .

Computer Technology Today

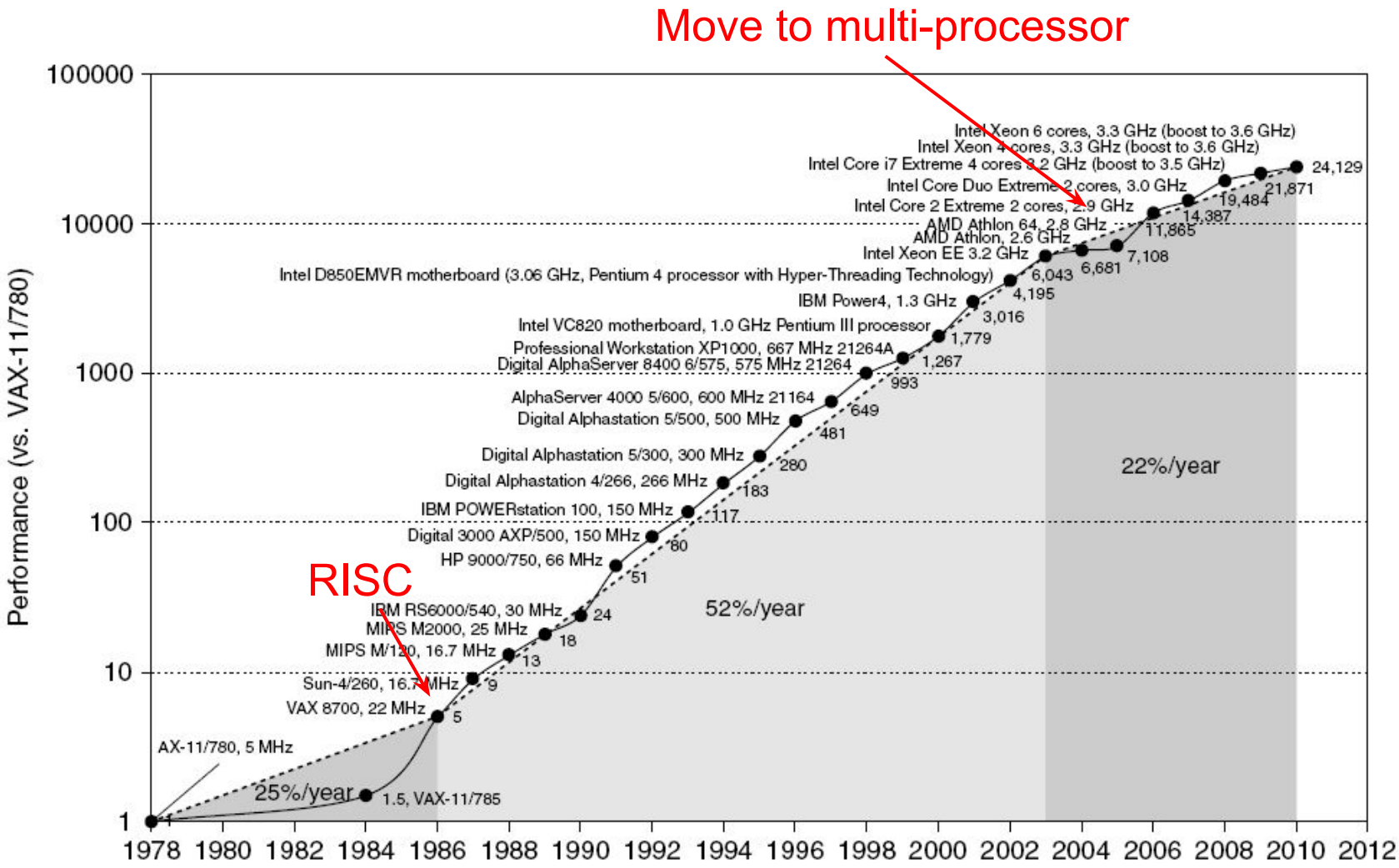
Today's Desk Top Computer for US$ 500 more powerful than a

Powerful : Performance, Main Memory & Disk Storage

Computer Technology

- Performance improvements:
  - Improvements in semiconductor technology
    - Feature size, clock speed
  - Improvements in computer architectures
    - Enabled by HLL compilers, UNIX
    - Lead to RISC architectures
  - Together have enabled:
    - Lightweight computers
    - Productivity-based managed/interpreted programming languages

Single Processor Performance



Current Trends in Architecture

- Cannot continue to leverage Instruction-Level parallelism (ILP)
  - Single processor performance improvement ended in 2003
- New models for performance:
  - Data-level parallelism (DLP)
  - Thread-level parallelism (TLP)
  - Request-level parallelism (RLP)
- These require explicit restructuring of the application

Parallelism

- Classes of parallelism in applications:
  - Data-Level Parallelism (DLP)
  - Task-Level Parallelism (TLP)
- Classes of architectural parallelism:
  - Instruction-Level Parallelism (ILP)
  - Vector architectures/Graphic Processor Units (GPUs)
  - Thread-Level Parallelism
  - Request-Level Parallelism

Parallelism

- 1. Instruction-Level Parallelism exploits DLP eg. pipelining and speculative execution.
- 2. Vector Architectures and Graphic Processor Units (GPUs) exploit DLP by applying a single instruction to a collection of data in parallel.
- 3. Thread-Level Parallelism exploits either DLP or TLP in a tightly coupled hardware model that allows for interaction among parallel threads.
- 4. Request-Level Parallelism exploits parallelism among largely decoupled tasks specified by the programmer or the operating system.

Flynn's Taxonomy

- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data streams (SIMD)
  - Vector architectures
  - Multimedia extensions
  - Graphics processor units
- Multiple instruction streams, single data stream (MISD)
  - No commercial implementation
- Multiple instruction streams, multiple data streams (MIMD)
  - Tightly-coupled MIMD
  - Loosely-coupled MIMD:Cluster Computing

Defining Computer Architecture

- "Old" view of computer architecture:
  - Instruction Set Architecture (ISA) design
  - i.e. decisions regarding:
    - registers, memory addressing, addressing modes, instruction operands, available operations, control flow instructions, instruction encoding
- "Real" computer architecture:
  - Specific requirements of the target machine
  - Design to maximize performance within constraints: cost, power, and availability
  - Includes ISA, microarchitecture, hardware

Technology

- If an ISA is to be successful, it must be designed to survive rapid changes in computer technology. as successful ISA may last decades—for example, the core of the IBM mainframe has been in use for nearly 50 years.
- An architect must plan for technology changes that can increase the lifetime of a successful computer.
- To plan for the evolution of a computer, the designer must be aware of rapid changes in implementation technology.
- Five implementation technologies, which change at a dramatic pace, are critical to modern implementations:

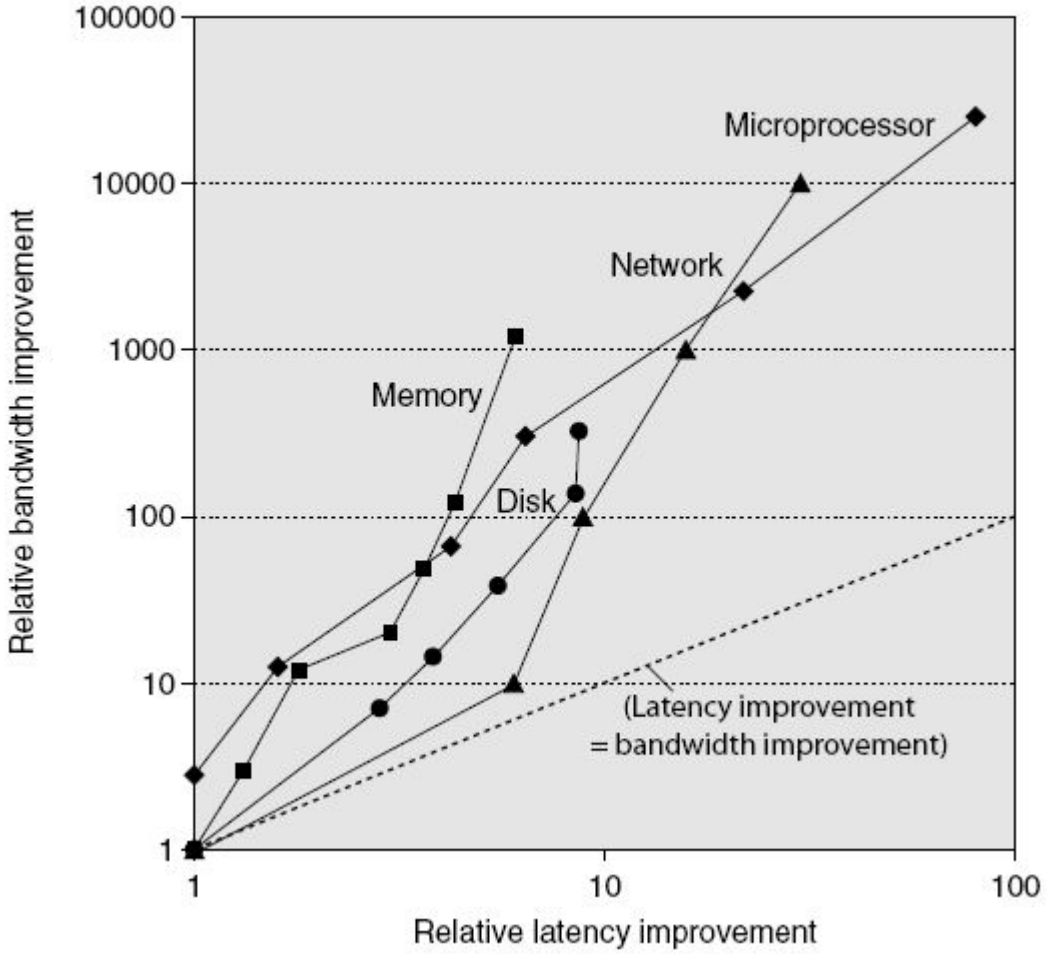
Trends in Technology

- Integrated circuit technology
  - Transistor density: 35%/year
  - Die size: 10-20%/year
  - Integration overall: 40-55%/year
- DRAM capacity: 25-40%/year (slowing)
- Flash capacity: 50-60%/year
  - 15-20X cheaper/bit than DRAM
- Magnetic disk technology: 40%/year
  - 15-25X cheaper/bit then Flash
  - 300-500X cheaper/bit than DRAM

Bandwidth and Latency

- Bandwidth or throughput
  - Total work done in a given time
  - 10,000-25,000X improvement for processors
  - 300-1200X improvement for memory and disks
- Latency or response time
  - Time between start and completion of an event
  - 30-80X improvement for processors
  - 6-8X improvement for memory and disks

Bandwidth and Latency



Log-log plot of bandwidth and latency milestones

Transistors and Wires

- Feature size
  - Minimum size of transistor or wire in x or y dimension
  - 10 microns in 1971 to .032 microns in 2011
  - Transistor performance scales linearly
    - Wire delay does not improve with feature size!
  - Integration density scales quadratically

Power and Energy

- Problem: Get power in, get power out
- Thermal Design Power (TDP)
  - Characterizes sustained power consumption
  - Used as target for power supply and cooling system
  - Lower than peak power, higher than average power consumption
- Clock rate can be reduced dynamically to limit power consumption
- Energy per task is often a better measurement

Dynamic Energy and Power

- Dynamic energy
  - Transistor switch from 0 -> 1 or 1 -> 0
  - ½ x Capacitive load x Voltage^2
- Dynamic power
  - ½ x Capacitive load x Voltage^2 x Frequency switched
- Reducing clock rate reduces power, not energy
- For example, processor A may have a 20% higher average power consumption than processor B, but if A executes the task in only 70% of the time needed by B, its energy consumption will be 1.2 × 0.7 = 0.84, which is clearly better.

Dynamic Energy and Power

Example Some microprocessors today are designed to have adjustable voltage, so a 15% reduction in voltage may result in a 15% reduction in frequency. What would be the impact on dynamic energy and on dynamic power?

Answer Since the capacitance is unchanged, the answer for energy is the ratio of the voltages since the capacitance is unchanged:

$$\frac{\text{Energy}_{\text{new}}}{\text{Energy}_{\text{old}}} = \frac{(\text{Voltage} \times 0.85)^2}{\text{Voltage}^2} = 0.85^2 = 0.72$$

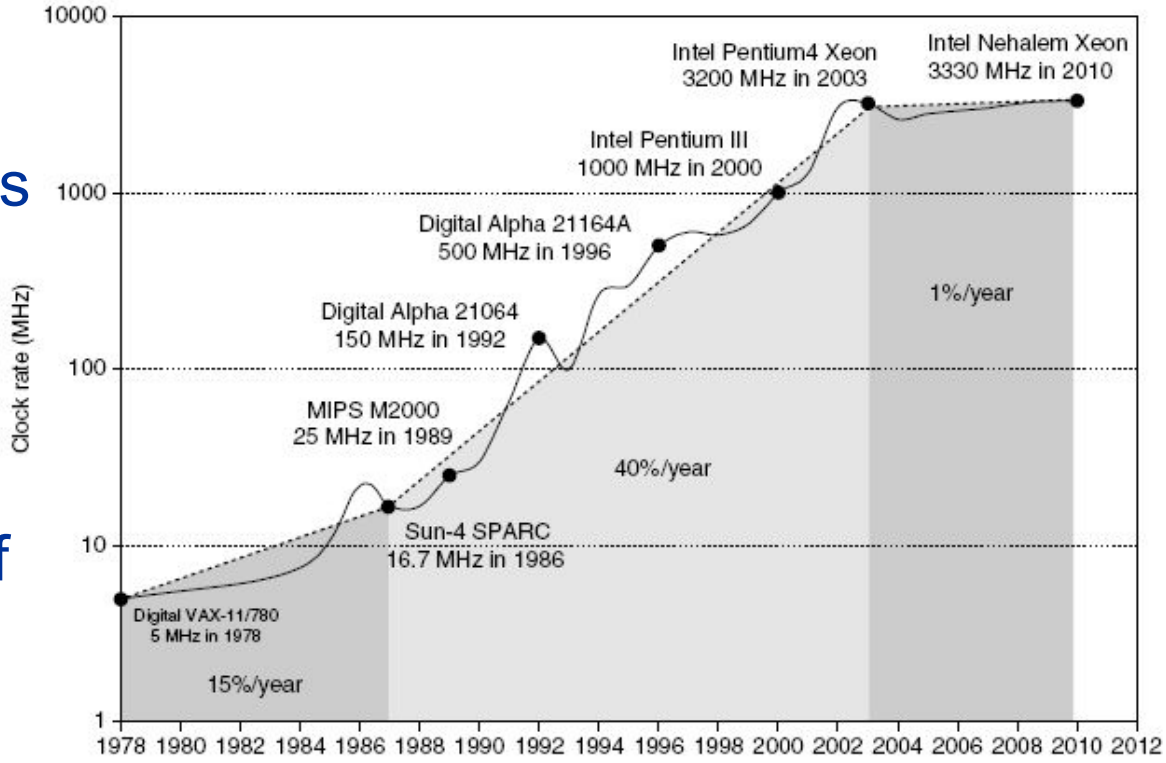
thereby reducing energy to about 72% of the original. For power, we add the ratio of the frequencies

$$\frac{\text{Power}_{\text{new}}}{\text{Power}_{\text{old}}} = 0.72 \times \frac{(\text{Frequency switched} \times 0.85)}{\text{Frequency switched}} = 0.61$$

shrinking power to about 61% of the original.

Power

- Intel 80386 consumed ~ 2 W
- 3.3 GHz Intel Core i7 consumes 130 W
- Heat must be dissipated from 1.5 x 1.5 cm chip
- This is the limit of what can be cooled by air



Reducing Power

- Techniques for reducing power:
  - Do nothing well
    - Most microprocessors today turn off the clock of inactive modules to save energy and dynamic power. For example, if no floating-point instructions are executing, the clock of the floating-point unit is disabled.
  - Dynamic Voltage-Frequency Scaling:
    - periods of low activity where there is no need to operate at the highest clock frequency and voltages. Modern microprocessors typically offer a few clock frequencies and voltages in which to operate that use lower power and energy

- Low power state for DRAM, disks:
- low power modes to save energy

- Overclocking, turning off cores:

Turbo mode: if temperature rises beyond certain threshold: turnoff few core: performance vary wrt time.

Static Power

- Static power consumption
  - $Current_{static}$ x Voltage
  - Scales with number of transistors
  - To reduce: power gating
  - **Power gating** is a technique used in integrated circuit design to reduce**power** consumption, by shutting off the current to blocks of the circuit that are not in

Trends in Cost

- Cost driven down by learning curve
  - Yield
- DRAM: price closely tracks cost
- Microprocessors: price depends on volume
  - 10% less for each doubling of volume

Integrated Circuit Cost

- Integrated circuit

$$\text{Cost of integrated circuit} = \frac{\text{Cost of die} + \text{Cost of testing die} + \text{Cost of packaging and final test}}{\text{Final test yield}}$$

$$\text{Cost of die} = \frac{\text{Cost of wafer}}{\text{Dies per wafer} \times \text{Die yield}}$$

$$\text{Dies per wafer} = \frac{\pi \times (\text{Wafer diameter}/2)^2}{\text{Die area}} - \frac{\pi \times \text{Wafer diameter}}{\sqrt{2 \times \text{Die area}}}$$

- Bose-Einstein formula:

$$\text{Die yield} = \text{Wafer yield} \times 1/(1 + \text{Defects per unit area} \times \text{Die area})^N$$

- Defects per unit area = 0.016-0.057 defects per square cm (2010)
- N = process-complexity factor = 11.5-15.5 (40 nm, 2010)

Integrated Circuit Cost

Example Find the die yield for dies that are 1.5 cm on a side and 1.0 cm on a side, assuming a defect density of 0.031 per cm^2 and N is 13.5.

Answer The total die areas are 2.25 cm^2 and 1.00 cm^2. For the larger die, the yield is

$$\text{Die yield} = 1/(1 + 0.031 \times 2.25)^{13.5} = 0.40$$

For the smaller die, the yield is

$$\text{Die yield} = 1/(1 + 0.031 \times 1.00)^{13.5} = 0.66$$

That is, less than half of all the large dies are good but two-thirds of the small dies are good.

Integrated Circuit Cost

Example Find the number of dies per 300 mm (30 cm) wafer for a die that is 1.5 cm on a side and for a die that is 1.0 cm on a side.

Answer When die area is 2.25 cm^2:

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{2.25} - \frac{\pi \times 30}{\sqrt{2 \times 2.25}} = \frac{706.9}{2.25} - \frac{94.2}{2.12} = 270$$

Since the area of the larger die is 2.25 times bigger, there are roughly 2.25 as many smaller dies per wafer:

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{1.00} - \frac{\pi \times 30}{\sqrt{2 \times 1.00}} = \frac{706.9}{1.00} - \frac{94.2}{1.41} = 640$$

Dependability

- Module reliability
  - Mean time to failure (MTTF)
  - Mean time to repair (MTTR)
  - Mean time between failures (MTBF) = MTTF + MTTR
  - Availability = MTTF / MTBF

Dependability

Example Assume a disk subsystem with the following components and MTTF:

- 10 disks, each rated at 1,000,000-hour MTTF
- 1 ATA controller, 500,000-hour MTTF
- 1 power supply, 200,000-hour MTTF
- 1 fan, 200,000-hour MTTF
- 1 ATA cable, 1,000,000-hour MTTF

Using the simplifying assumptions that the lifetimes are exponentially distributed and that failures are independent, compute the MTTF of the system as a whole.

Answer The sum of the failure rates is

$$\text{Failure rate}_{\text{system}} = 10 \times \frac{1}{1{,}000{,}000} + \frac{1}{500{,}000} + \frac{1}{200{,}000} + \frac{1}{200{,}000} + \frac{1}{1{,}000{,}000}$$

$$= \frac{10 + 2 + 5 + 5 + 1}{1{,}000{,}000 \text{ hours}} = \frac{23}{1{,}000{,}000} = \frac{23{,}000}{1{,}000{,}000{,}000 \text{ hours}}$$

or 23,000 FIT. The MTTF for the system is just the inverse of the failure rate:

$$\text{MTTF}_{\text{system}} = \frac{1}{\text{Failure rate}_{\text{system}}} = \frac{1{,}000{,}000{,}000 \text{ hours}}{23{,}000} = 43{,}500 \text{ hours}$$

or just under 5 years.

Measuring Performance

- Typical performance metrics:
  - Response time
  - Throughput
- Speedup of X relative to Y
  - Execution time$_Y$ / Execution time$_X$
- Execution time
  - Wall clock time: includes all system overheads
  - CPU time: only computation time
- Benchmarks
  - Kernels (e.g. matrix multiply)
  - Toy programs (e.g. sorting)
  - Synthetic benchmarks (e.g. Dhrystone)
  - Benchmark suites (e.g. SPEC06fp, TPC-C)

Principles of Computer Design

- Take Advantage of Parallelism
  - e.g. multiple processors, disks, memory banks, pipelining, multiple functional units
- Principle of Locality
  - Reuse of data and instructions
  - Temporal locality states that recently accessed items are likely to be accessed in the near future.
  - Spatial locality says that items whose addresses are near one another tend to be referenced close together in time.

Amdahl's Law

- Speedup is defined as the time it takes a program to execute in serial (with one processor) divided by the time it takes to execute in parallel (with many processors). The formula for speedup is:

$$S = \frac{T(1)}{T(j)}$$

Where T(j) is the time it takes to execute the program when using j processors.

Amdahl's Law

- If there are N workers working on a project, we may assume that they would be able to do a job in 1/N time of one worker working alone.
- Now, if we assume the strictly serial part of the program is performed in B*T(1) time,
- then the strictly parallel part is performed in ((1-B)*T(1)) / N time. With some substitution and number manipulation, we get the formula for speedup as:

$$S = \frac{N}{(B*N)+(1-B)}$$

Amdahl's Law

Example Suppose that we want to enhance the processor used for Web serving. The new processor is 10 times faster on computation in the Web serving application than the original processor. Assuming that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

Answer $\text{Fraction}_{\text{enhanced}} = 0.4$; $\text{Speedup}_{\text{enhanced}} = 10$; $\text{Speedup}_{\text{overall}} = \dfrac{1}{0.6 + \dfrac{0.4}{10}} = \dfrac{1}{0.64} = 1.56$

Amdahl's Law

Example A common transformation required in graphics processors is square root. Implementations of floating-point (FP) square root vary significantly in performance, especially among processors designed for graphics. Suppose FP square root (FPSQR) is responsible for 20% of the execution time of a critical graphics benchmark. One proposal is to enhance the FPSQR hardware and speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions in the graphics processor run faster by a factor of 1.6; FP instructions are responsible for half of the execution time for the application. The design team believes that they can make all FP instructions run 1.6 times faster with the same effort as required for the fast square root. Compare these two design alternatives.

Amdahl's Law

Answer We can compare these two alternatives by comparing the speedups:

$$\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

Improving the performance of the FP operations overall is slightly better because of the higher frequency.

Principles of Computer Design

- Focus on the Common Case
  - Amdahl's Law

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Principles of Computer Design

- The Processor Performance Equation

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

$$\text{CPU time} = \text{Instruction count} \times \text{Cycles per instruction} \times \text{Clock cycle time}$$

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

Principles of Computer Design

- Different instruction types having different CPIs

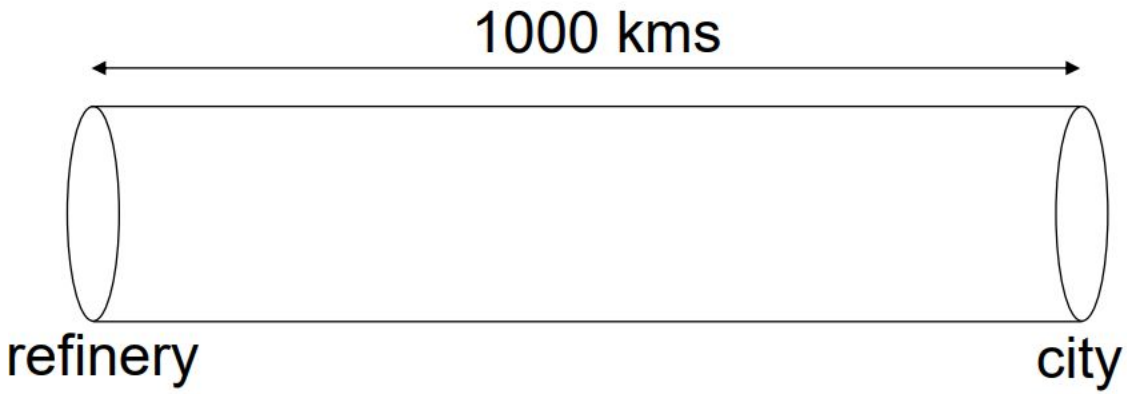
$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

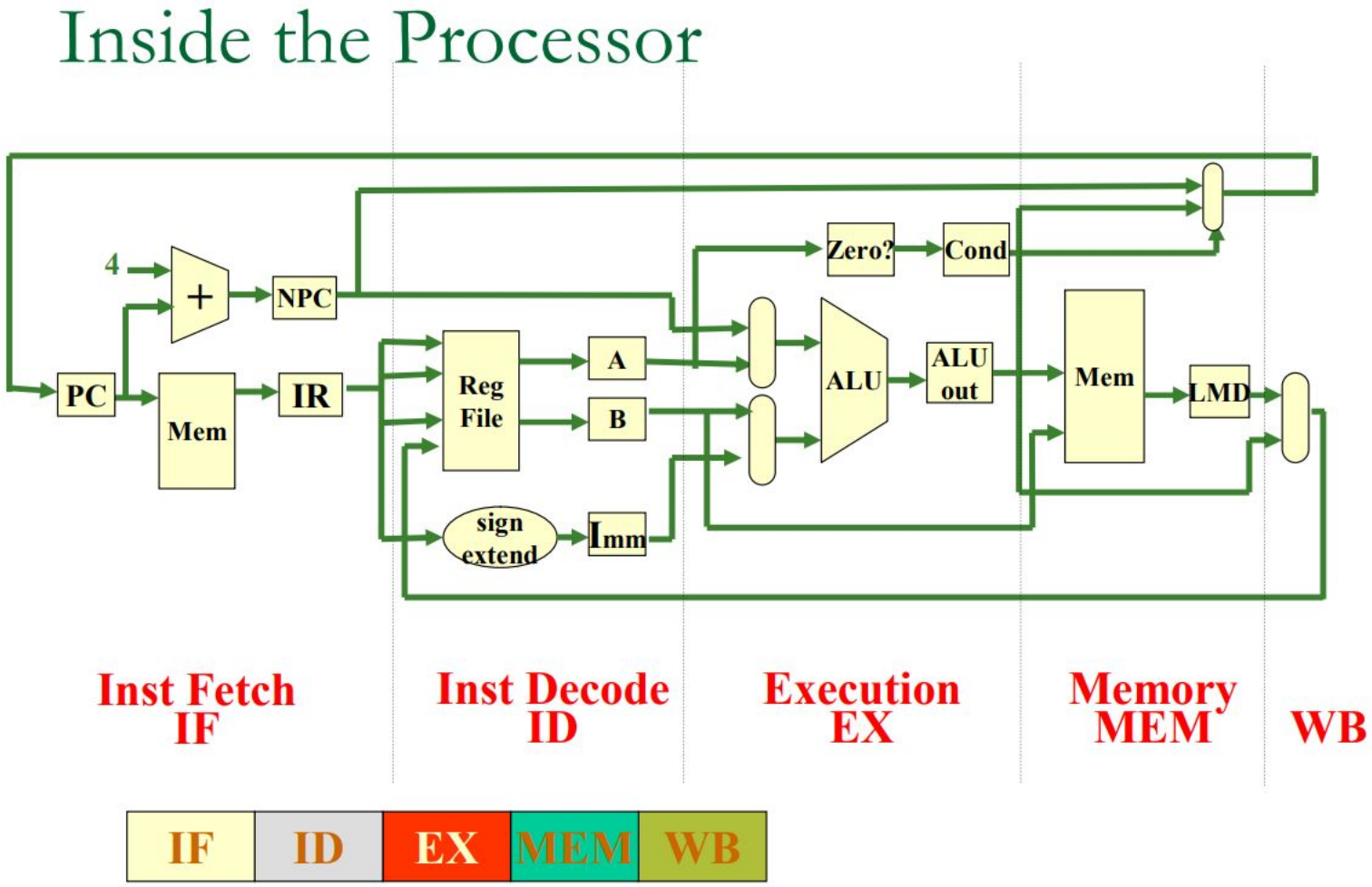
Pipelining

Pipelines

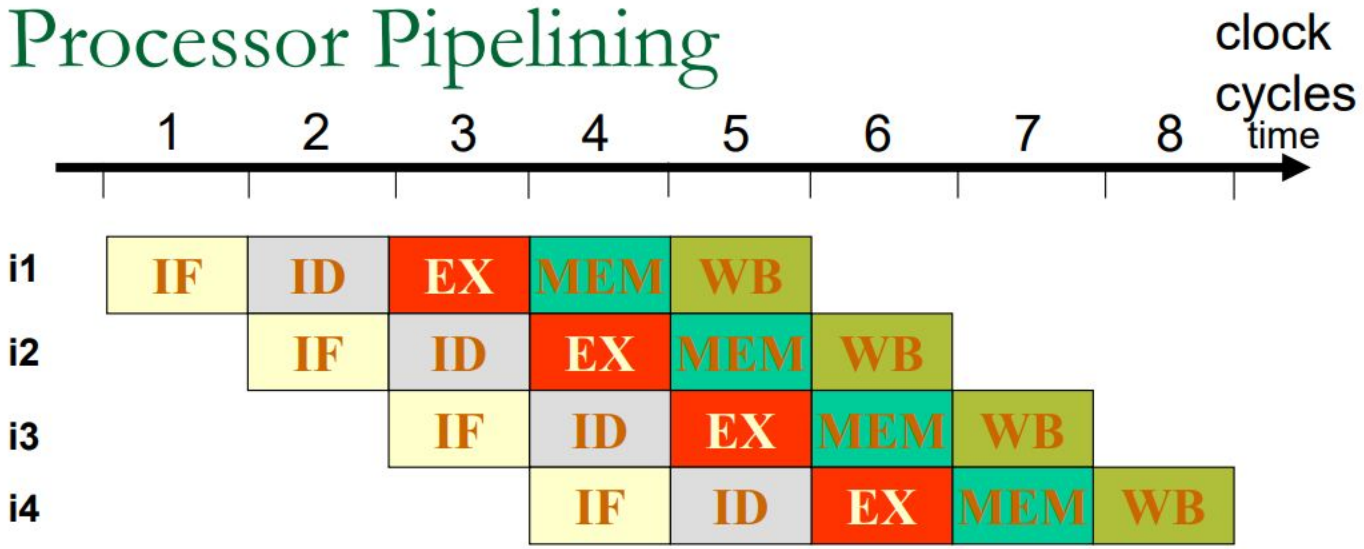
- Used for transportation of liquids or gases over long distances
  - 1000s of kms
  - Built with periodic pump/compressor stations to keep the fluid flowing



Pipelining

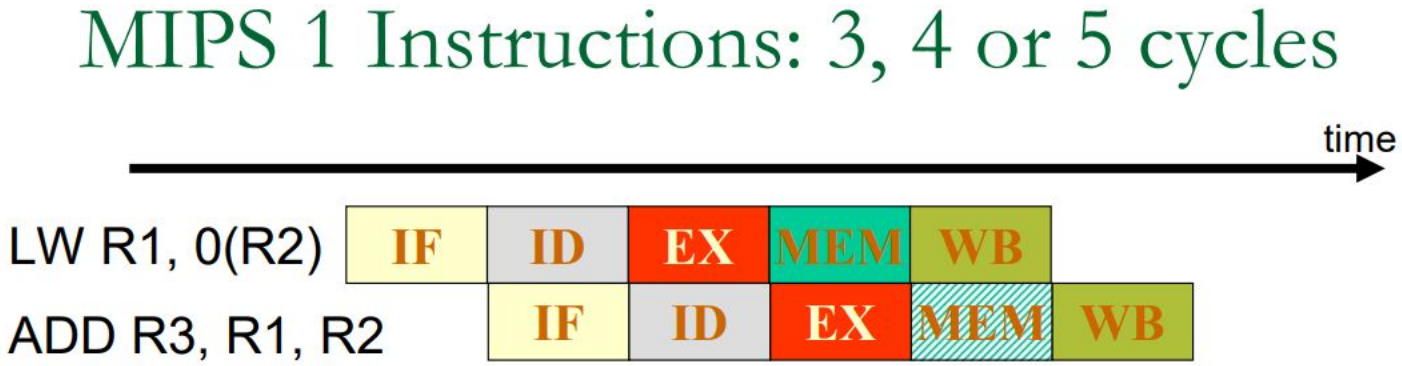


Pipelining



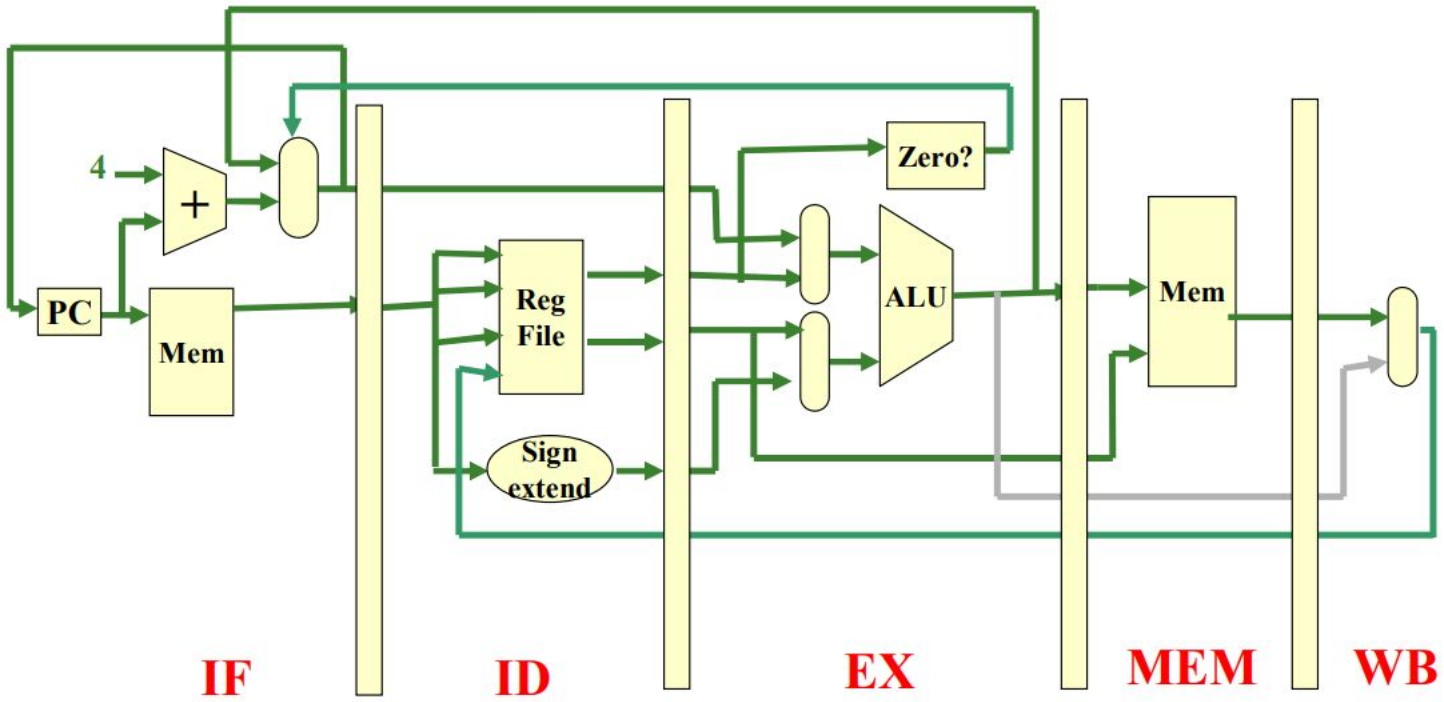
- Execution time of each instruction is still 5 cycles, but the throughput is now 1 instruction per cycle
- Initial pipeline fill time (4 cycles), after which 1 instruction completes every cycle

Pipelining



Pipelining

Pipelined Processor Datapath



Pipelining

Some Terminology

- Pipeline stages: IF, ID, EX, MEM, WB
- We describe this as a 5 stage pipeline
  - or a pipeline of depth 5
- Assume that the time delay through each stage is the same (say 1 clock cycle)

- Pipeline Speedup = $\frac{time_{non-pipelined}}{time_{pipelined}}$

Pipelining

Pipeline Speedup

- For 5 stage pipeline taking 1 cycle per stage
  - Let us compute the speedup over a non-pipelined processor that takes 5 cycles for every instruction
  - Calculate how much time each of these processors takes to run a program involving the execution of n instructions
    - Non-pipelined processor: $5n$ cycles
    - Pipelined processor: $4 + n$ cycles
  - Speedup = $\frac{5n}{4+n} \rightarrow 5 \quad \text{as} \quad n \rightarrow \infty$

Pipelining

Pipeline Speedup

- A pipeline with p stages could give a speedup of p (compared to a non-pipelined processor that takes p cycles for each instruction)
- i.e., A program would run p times faster on the pipelined processor (than on the non-pipelined processor)
  - if on every clock cycle, an instruction completes execution

Pipelining

Problem: Pipeline Hazards

A situation where an instruction cannot proceed through the pipeline as it should

Hazard: a dangerous (hazardous) situation

From the perspective of correct program execution

Pipelining

Problem: Pipeline Hazards

A situation where an instruction cannot proceed through the pipeline as it should

1. Structural hazard: When 2 or more instructions in the pipeline need to use the same resource at the same time
2. Data hazard: When an instruction depends on the data result of a prior instruction that is still in the pipeline
3. Control hazard: A hazard that arises due to control transfer instructions

Pipelining

Problem: Pipeline Hazards

A situation where an instruction cannot proceed through the pipeline as it should

1. Structural hazard: When 2 or more instructions in the pipeline need to use the same resource at the same time
2. Data hazard: When an instruction depends on the data result of a prior instruction that is still in the pipeline
3. Control hazard: A hazard that arises due to control transfer instructions

Pipelining

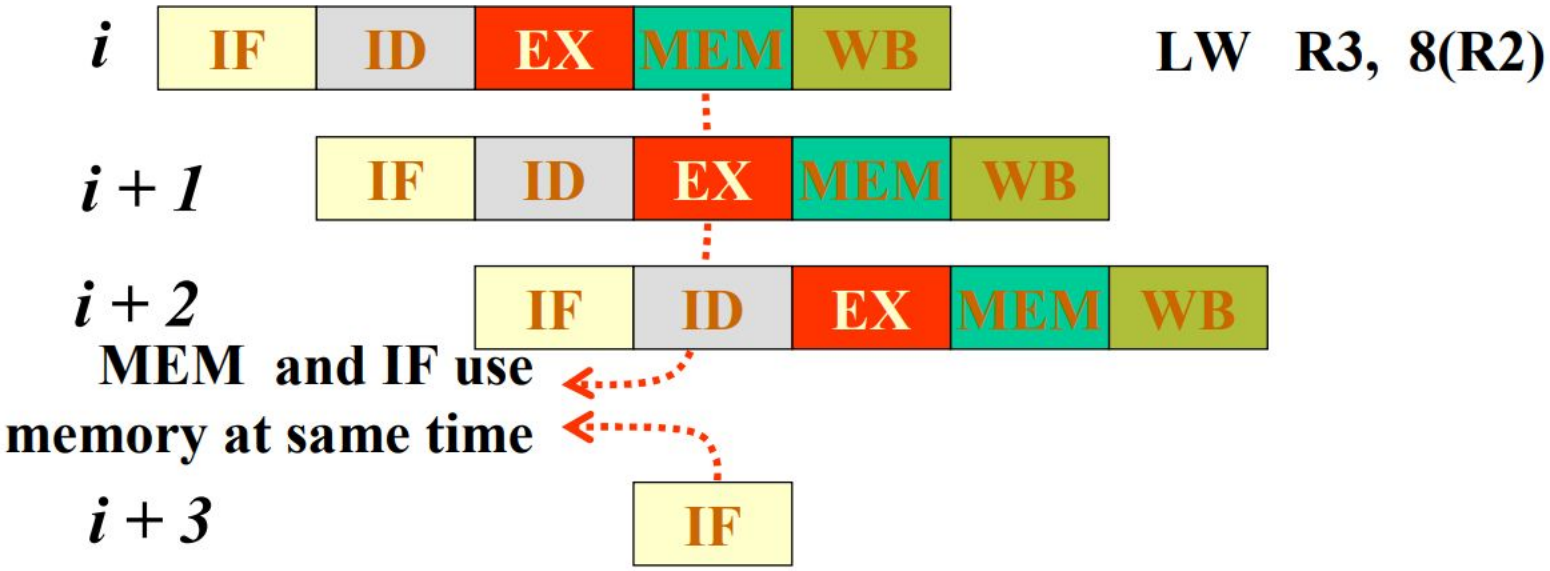
Problem: Pipeline Hazards

A situation where an instruction cannot proceed through the pipeline as it should

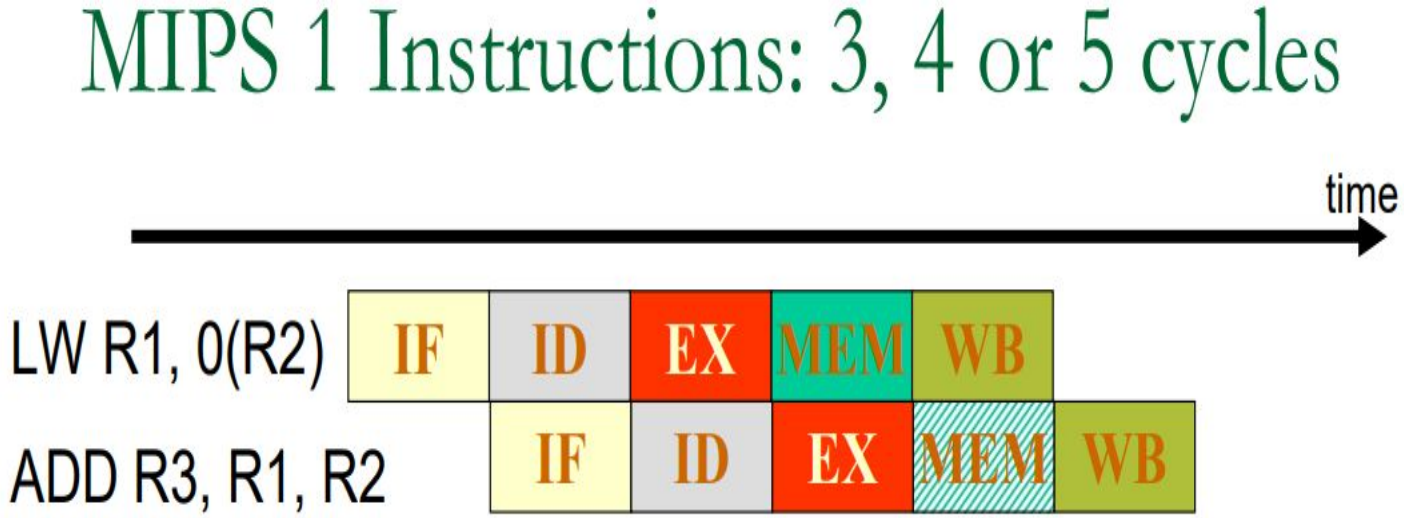
1. Structural hazard: When 2 or more instructions in the pipeline need to use the same resource at the same time
2. Data hazard: When an instruction depends on the data result of a prior instruction that is still in the pipeline
3. Control hazard: A hazard that arises due to control transfer instructions

Pipelining

Structural Hazard

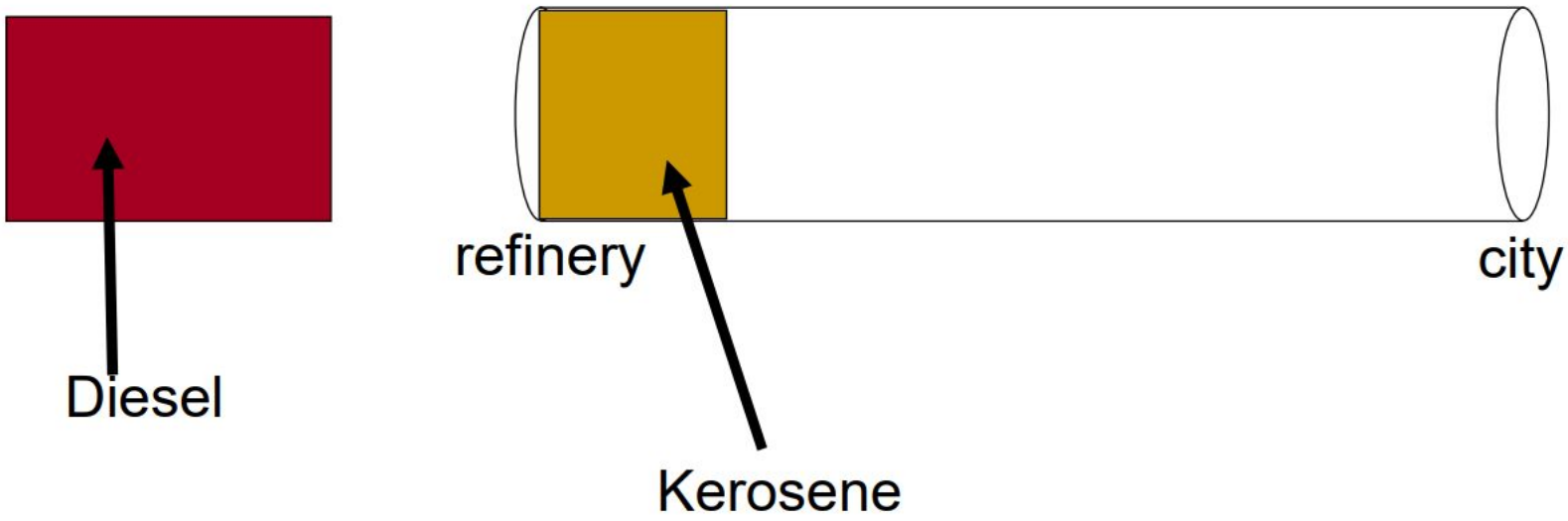


Pipelining



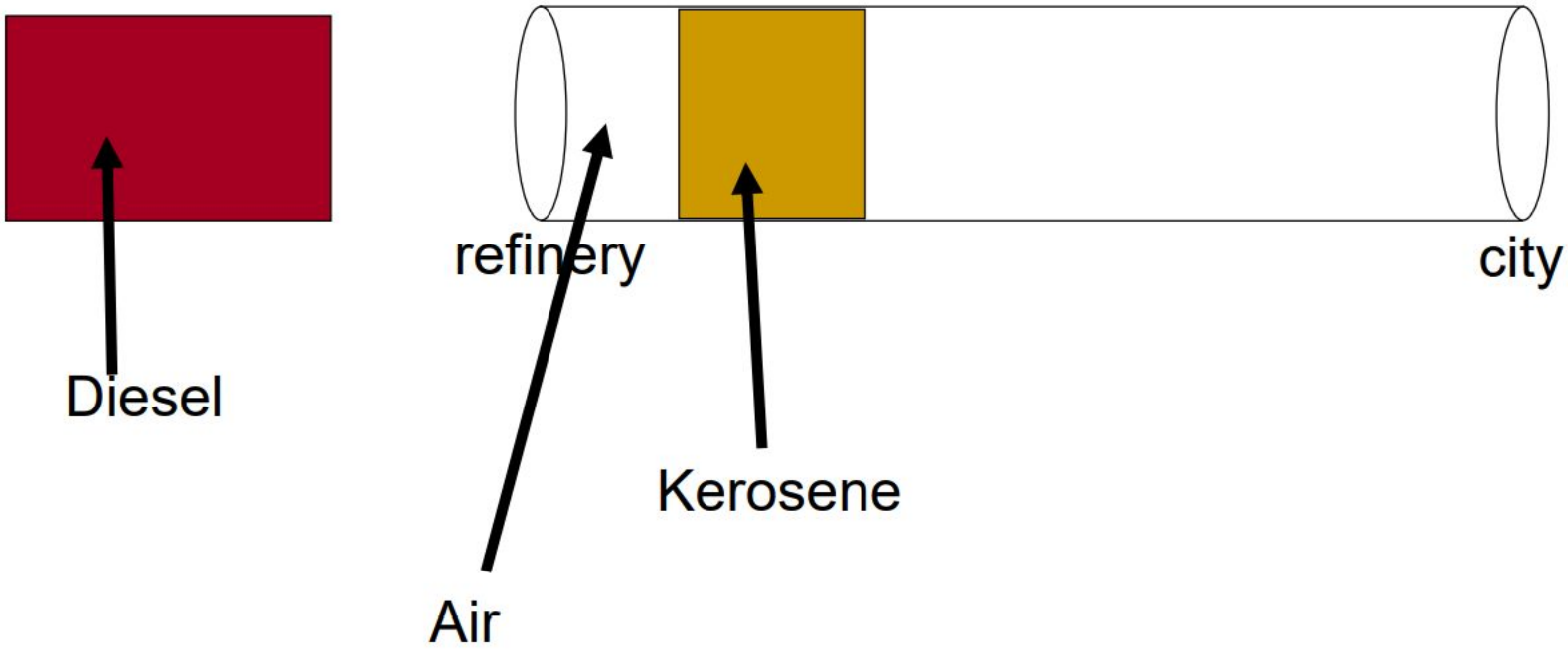
Pipelining

Petroleum pipeline analogy?

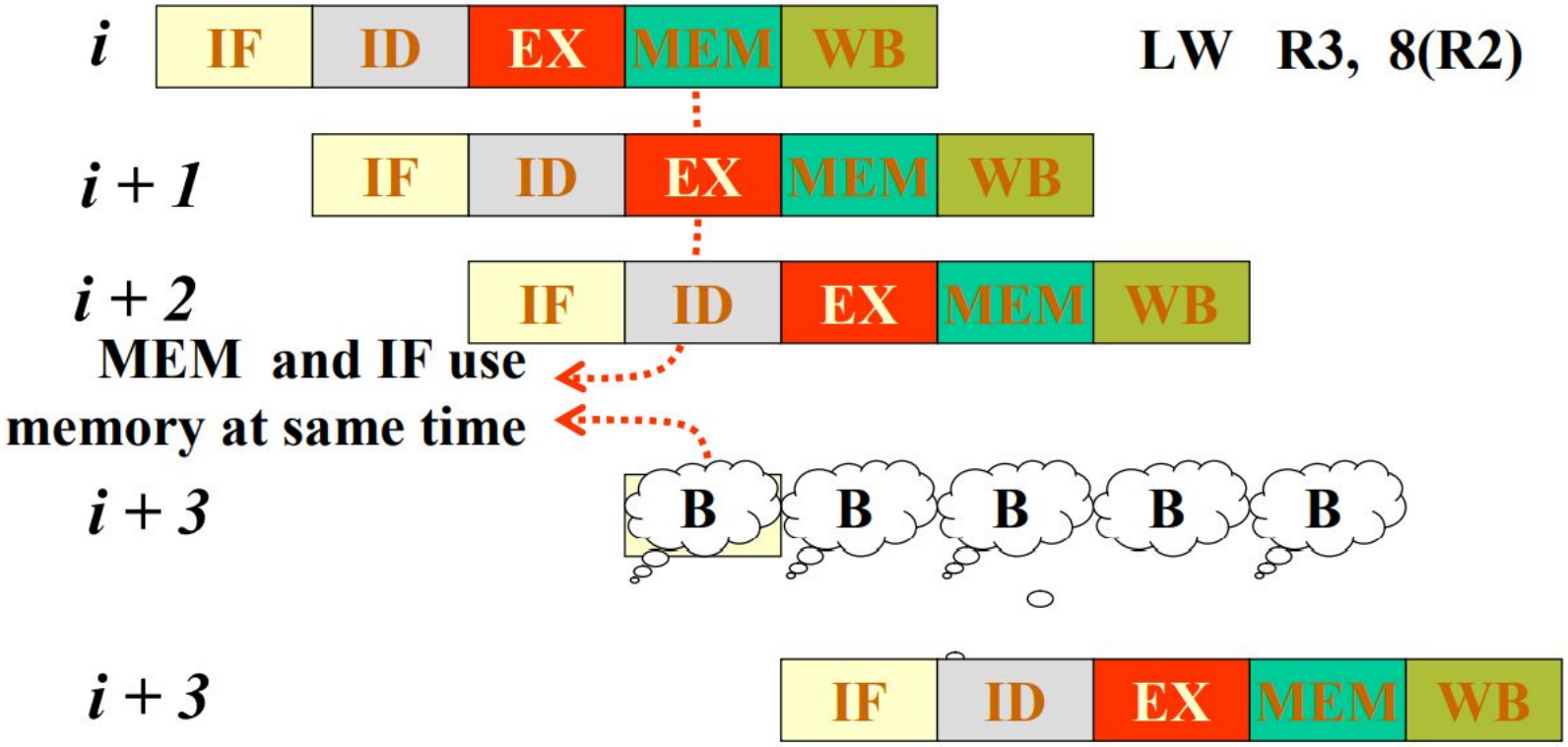


Pipelining

Petroleum pipeline analogy?



Pipelining



17

Pipelining

Solving Structural Hazards

- This hazard can be overcome by designing main memory so that it can handle 2 memory requests at the same time
  - Double ported memory
- Or, since we are assuming that memory delays are hidden by cache memories…
  - include a separate instruction cache (for use by the IF pipeline stage) and data cache (for use by the MEM stage)
- Identify the possible structural hazards and design so as to eliminate them

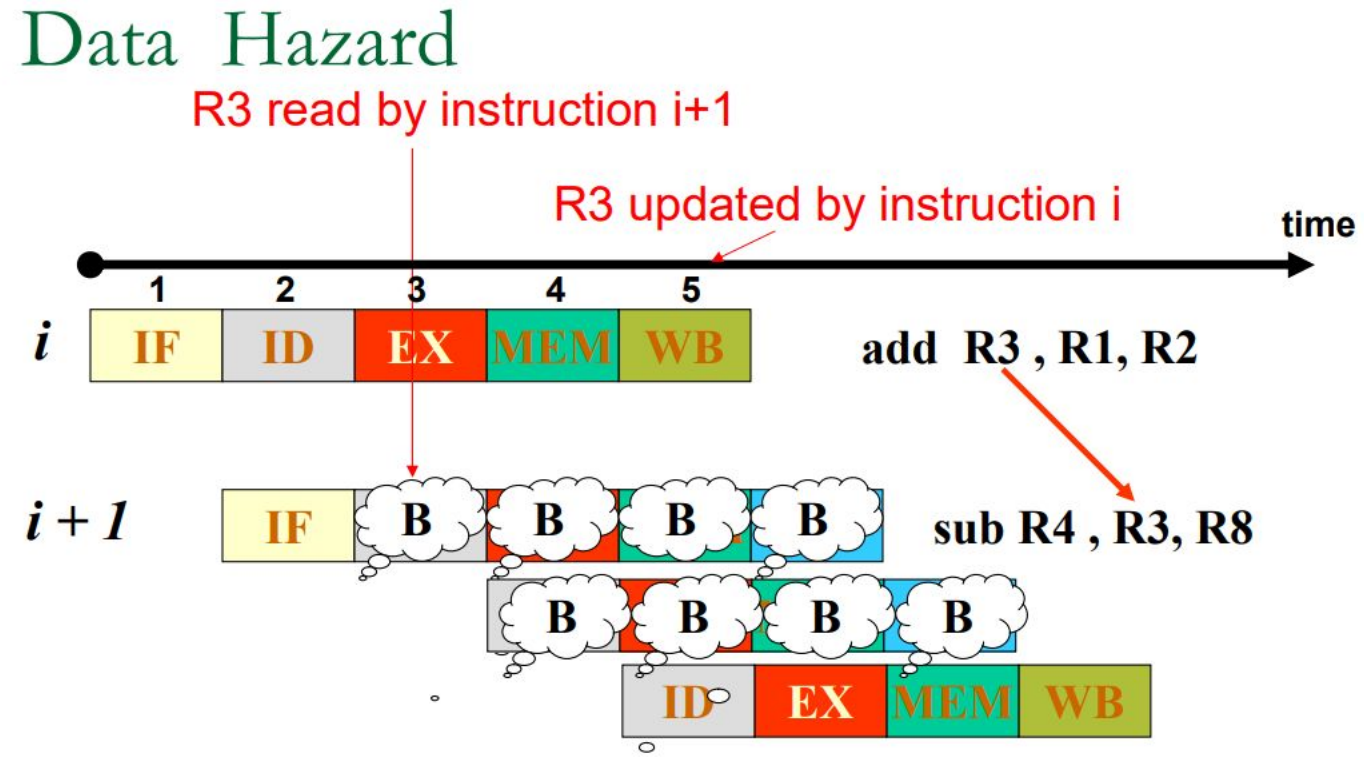
Pipelining

Problem: Pipeline Hazards

A situation where an instruction cannot proceed through the pipeline as it should

1. Structural hazard: When 2 or more instructions in the pipeline need to use the same resource at the same time
2. Data hazard: When an instruction depends on the data result of a prior instruction that is still in the pipeline
3. Control hazard: A hazard that arises due to control transfer instructions

Pipelining

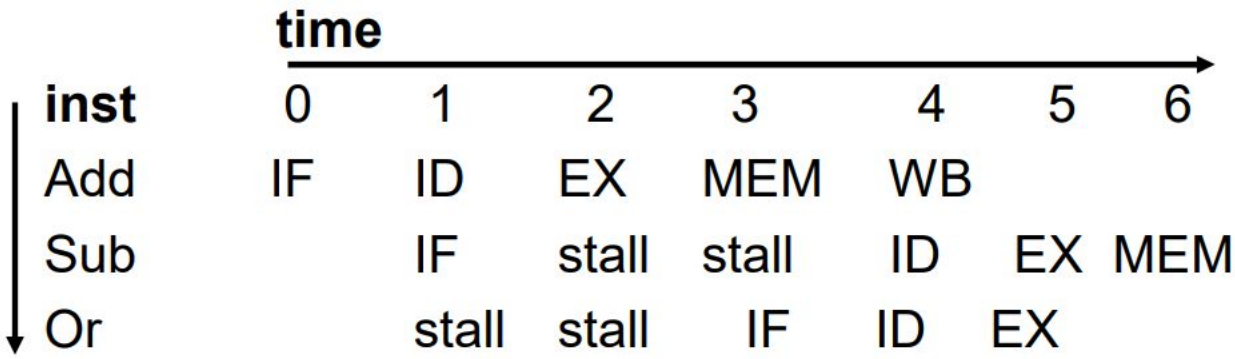


Idea: Delay (or stall) the progress of instruction i+1 through the pipeline until the data is available in register R3

Pipelining

Solving Data Hazards

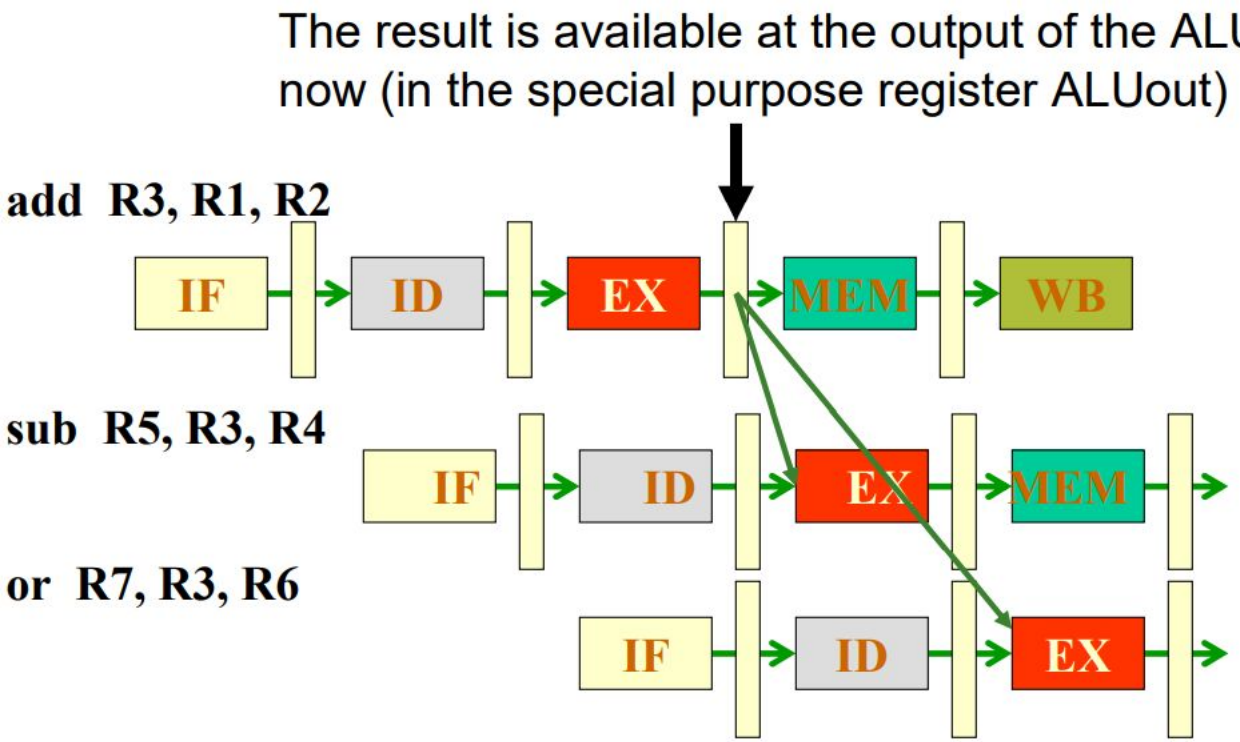
1. Interlock: Hardware that is included in the processor to detect such a data dependency and stall the dependent instruction



Pipelining

Solving Data Hazards

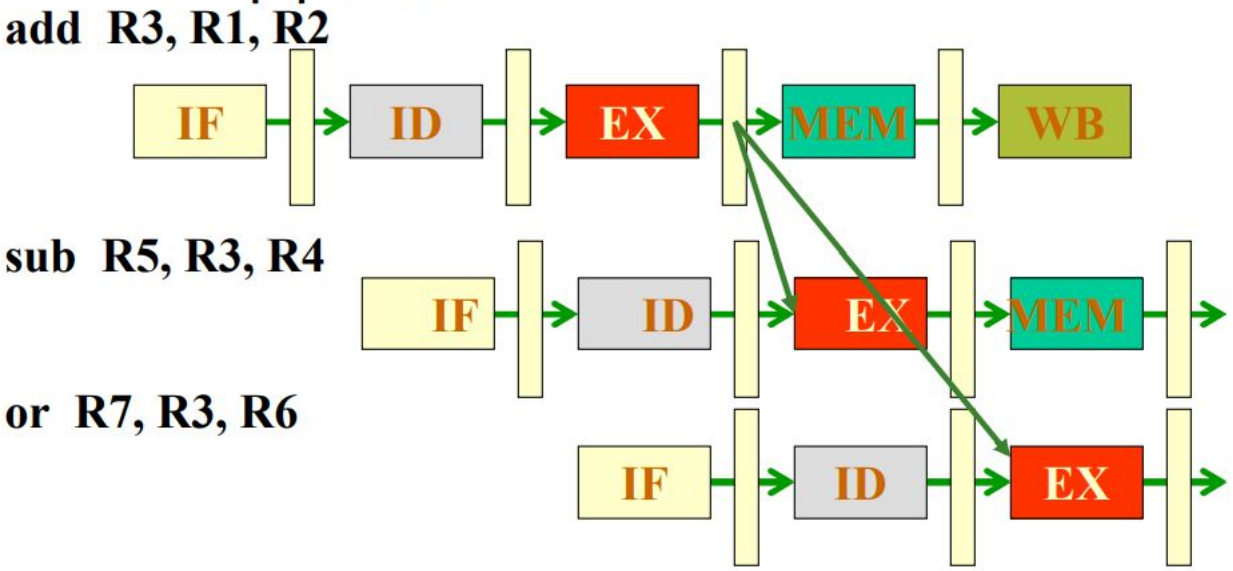
1. Interlocks & stalling dependent instructions



Pipelining

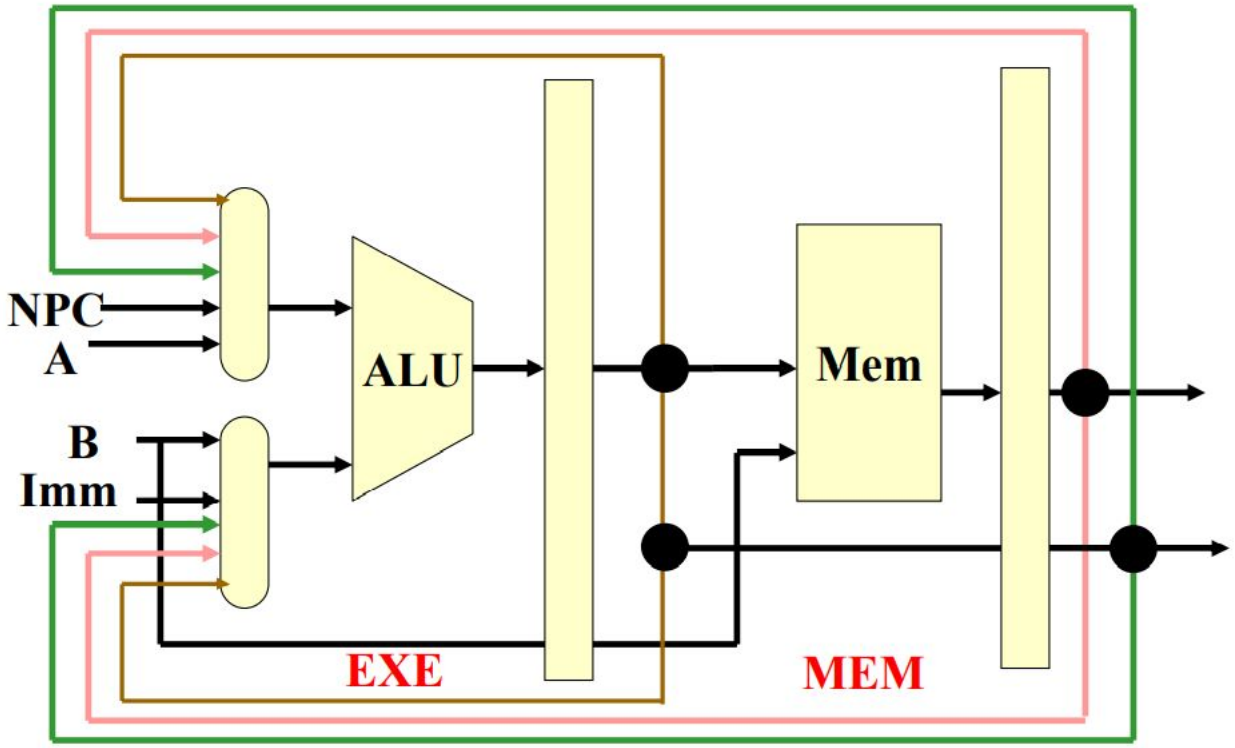
Solving Data Hazards

1. Interlocks & stalling dependent instructions
2. Forwarding or Bypassing: forward the result to EX as soon as is is available anywhere in the pipeline



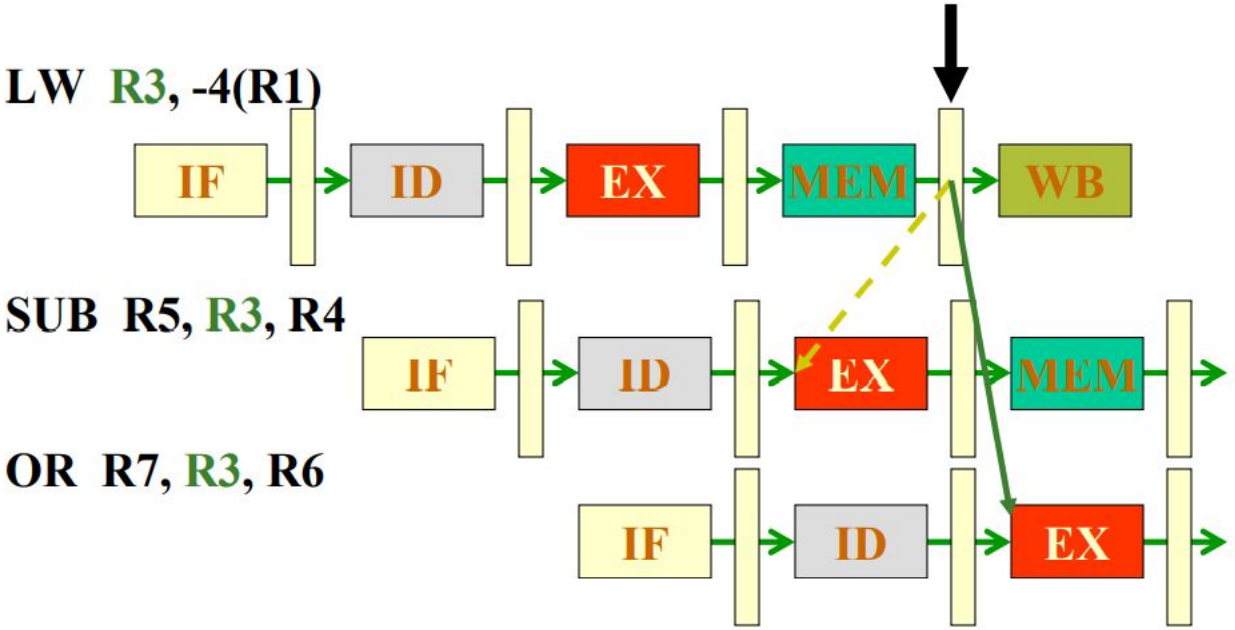
Pipelining

Modified Processor Datapath



Pipelining

But Forwarding is Not Always Possible



Pipelining

Solving Data Hazards

1. Interlocks & stalling dependent instructions
2. Forwarding or Bypassing
3. Load delay slot
   - Build the hardware to assume that an instruction that uses a load value is separated from the load instruction

Pipelining

Recall: Notes from ISA Manual.

- For load instructions: the loaded value might not be available in the destination register for use by the instruction immediately following the load
  - LOAD DELAY SLOT
- For control transfer instructions: the transfer of control takes place only following the instruction immediately after the control transfer instruction
  - BRANCH DELAY SLOT

Pipelining

Solving Data Hazards

1. Interlocks & stalling dependent instructions
2. Forwarding or Bypassing
3. Load delay slot
4. Instruction Scheduling
   - Reorder the instructions of the program so that dependent instructions are far enough apart
   - This could be done either
     - by the compiler, before the program runs: Static Instruction Scheduling
     - by the hardware, when the program is running: Dynamic Instruction Scheduling

Pipelining

Static Instruction Scheduling

- Reorder the instructions of the program to eliminate data hazards …
  - or in general to reduce the execution time of the program
- Reordering must be safe

```
ADD   R1,  R2,  R3      /*  R1 = R2 + R3  */
SUB   R2,  R4,  R5      /*  R2 = R4 – R5  */
```

Pipelining

Static Instruction Scheduling

- Reorder the instructions of the program to eliminate data hazards …
  - or in general to reduce the execution time of the program
- Reordering must be safe
  - should not change the meaning of the program
- Two instructions can be exchanged if they are independent of each other

Pipelining

Example: Static Instruction Scheduling

Program fragment:

```
LW   R3,  0(R1)
                ⇨ 1 stall
ADDI R5, R3, 1
ADD  R2, R2, R3
LW   R13, 0(R11)
                ⇨ 1 stall
ADD  R12, R13, R3
```

2 stalls

Scheduling:

```
LW R3, 0(R1)
ADDI R5, R3, 1
ADD R2, R2, R3
LW R13, 0(R11)
ADD R12, R13, R3
```

0 stalls

Pipelining

Solving Data Hazards

1. Interlocks & stalling dependent instructions
2. Forwarding or Bypassing
3. Load delay slot
4. Instruction Scheduling
   - Reorder the instructions of the program so that dependent instructions are far enough apart
   - This could be done either
     - by the compiler, before the program runs: Static Instruction Scheduling
     - by the hardware, when the program is running: Dynamic Instruction Scheduling

Pipelining

Kinds of Data Dependence

- True dependence

```
ADD   R1,  R2,  R3
SUB   R4,  R1,  R5
```

- Anti-dependence

```
ADD   R1,  R2,  R3
SUB   R2,  R4,  R5
```

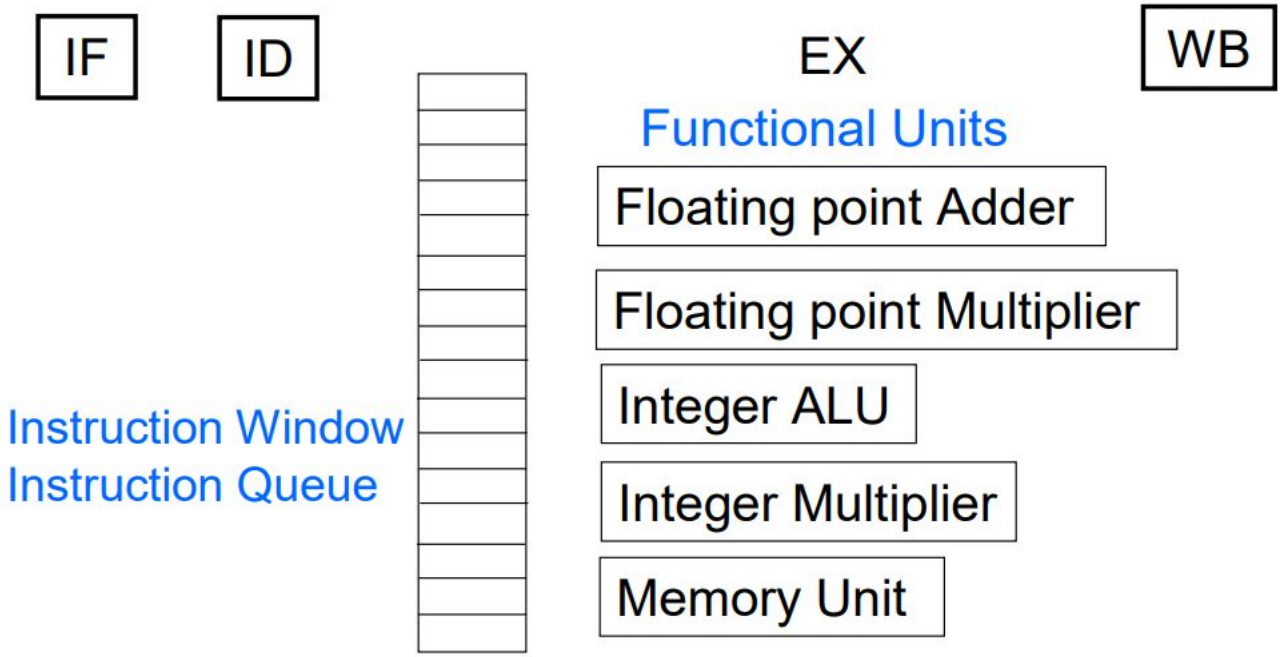
- Output dependence

```
ADD   R1,  R2,  R3
SUB   R1,  R4,  R5
```

Pipelining

Dynamic Instruction Scheduling

With dynamic instruction scheduling …



Pipelining

Dynamic Instruction Scheduling

- The hardware dynamically schedules instructions from the Instruction Window for execution on the functional units
- The instructions could execute in an order that is different from that specified by the program
  - with the same result
- Such processors are called “out of order” processors
  - as opposed to “in order” processors

Pipelining

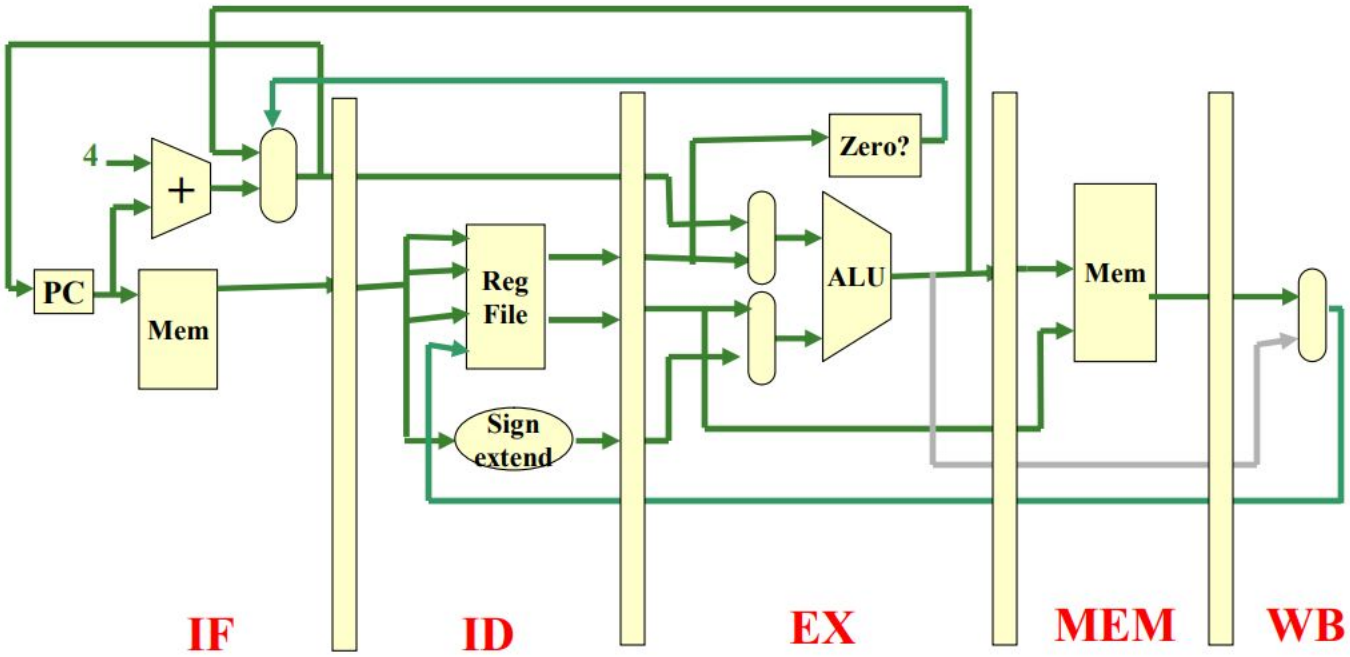
Problem: Pipeline Hazards

A situation where an instruction cannot proceed through the pipeline as it should

1. Structural hazard: When 2 or more instructions in the pipeline need to use the same resource at the same time
2. Data hazard: When an instruction depends on the data result of a prior instruction that is still in the pipeline
3. Control hazard: A hazard that arises due to control transfer instructions

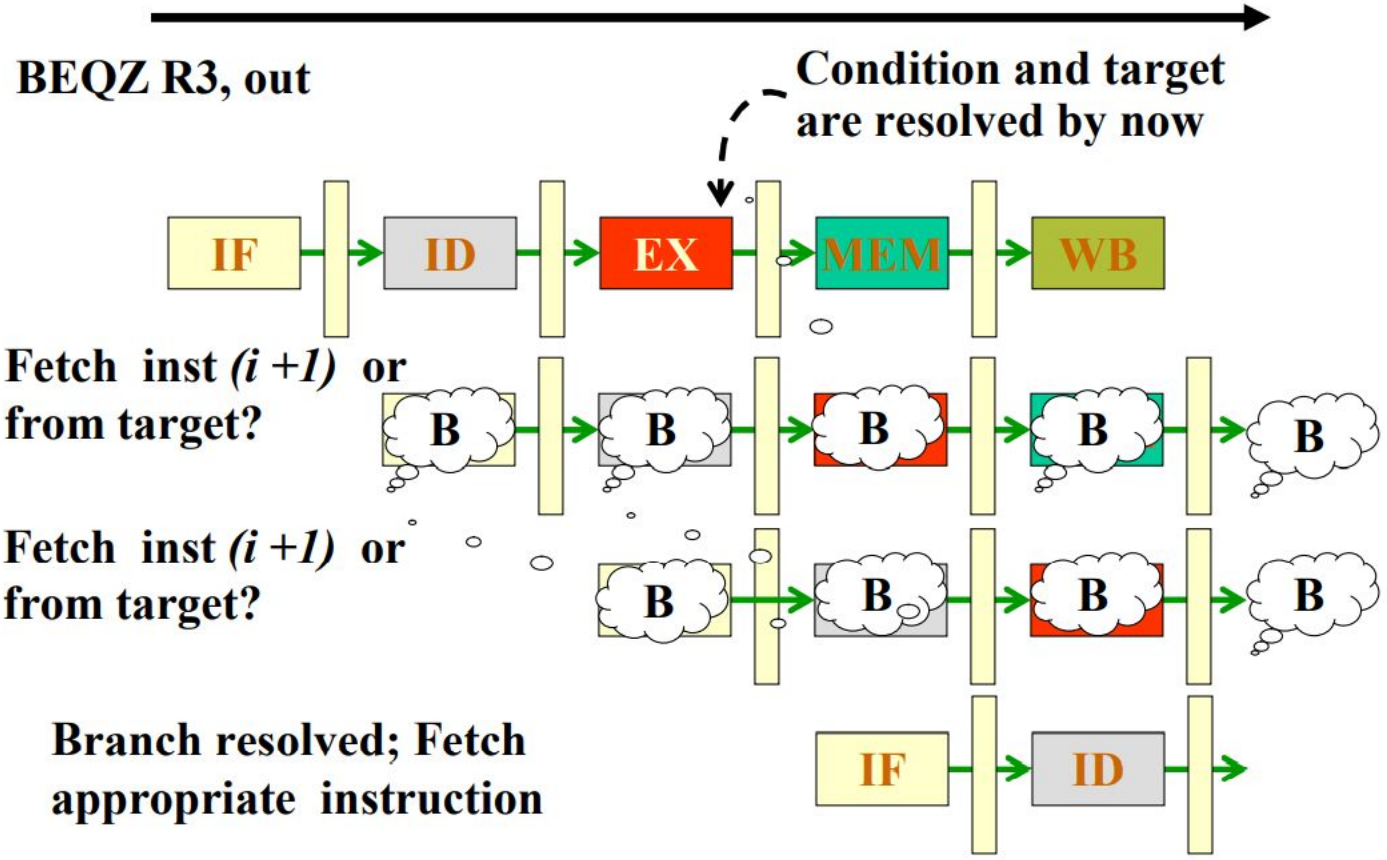
Pipelining

Recall: Execution of Branch Instruction



Pipelining

Control Hazards



Pipelining

Control Hazards

- Observation: Since the branch is resolved only in the EX stage, there must be 2 stall cycles after every conditional branch instruction

Pipelining

Reducing Impact of Branch Stall

- The execution of a conditional branch instruction involves 2 activities
  1. evaluating the branch condition (determine whether it is to be taken or not-taken)
  2. computing the branch target address
- To reduce branch stall effect we could
  - evaluate the condition earlier (in ID stage)
  - compute the target address earlier (in ID stage)
- The number of stall cycles would then be reduced to 1 cycle

Pipelining

Control Hazard Solutions

1. Static Branch Prediction

   Prediction?

   reasoning about the future

   guessing what is going to happen

   Static

   The behaviour of a branch instruction is predicted once before the program starts executing

Pipelining

Prediction and Correctness

- Prediction: guessing what is going to happen
- What if the guess is incorrect?
  - The pipelined processor hardware must be built to detect the misprediction and take appropriate corrective action

Pipelining

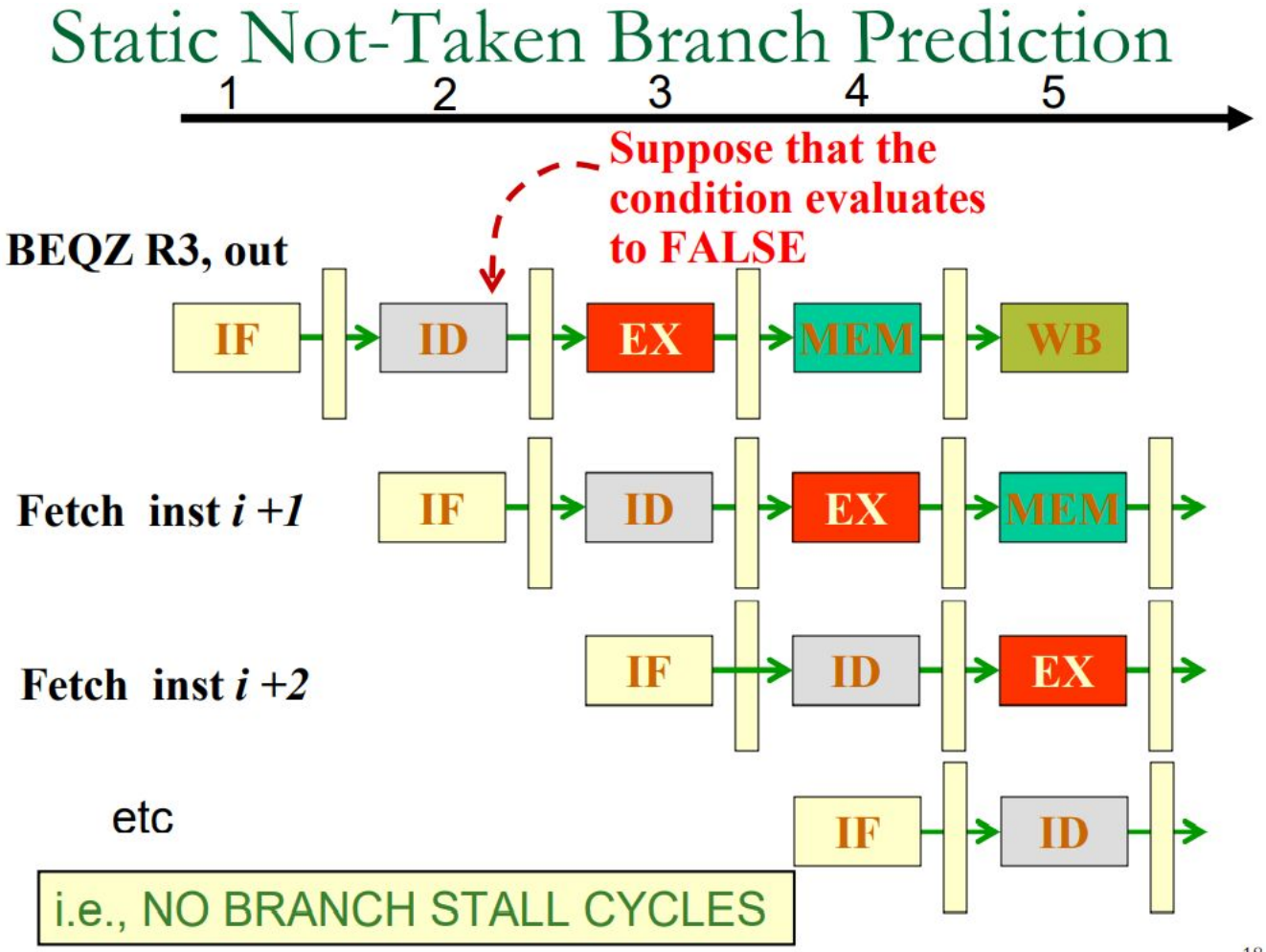
Control Hazard Solutions

1. Static Branch Prediction

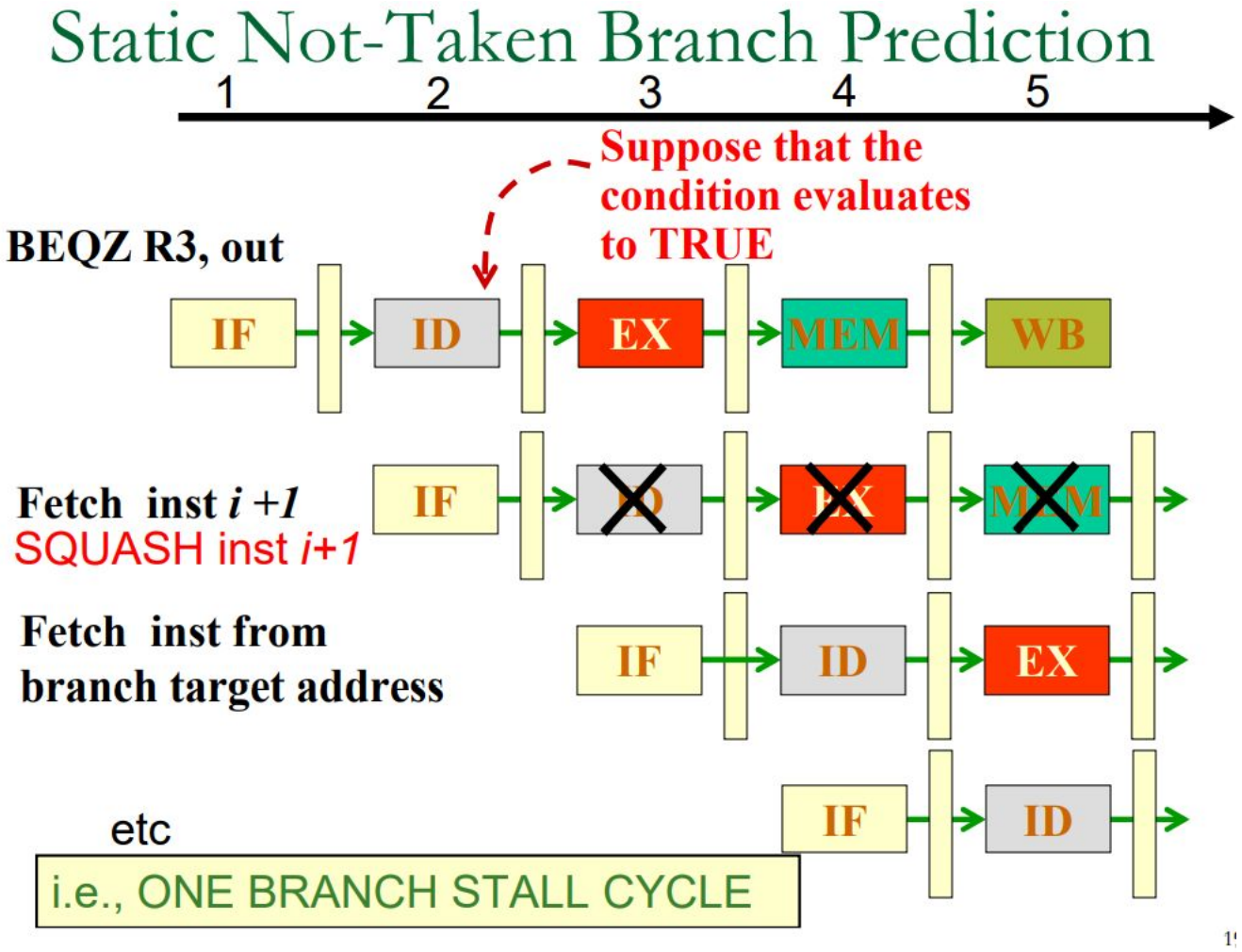
   Example: Static Not-Taken policy

   - The hardware is built to fetch next from PC + 4
   - After ID stage, if it is found that the branch condition is false (i.e., not taken), continue with the fetched instruction (from PC + 4)
     - Else, squash the fetched instruction and re-fetch from the branch target address
       - squash: cancel, annul the processing of that instruction

Pipelining



Pipelining



Pipelining

Control Hazard Solutions

1. Static Branch Prediction

   Example: Static Not-Taken policy

   - The hardware is built to fetch next from PC + 4
   - After ID stage, if it is found that the branch condition is false (i.e., not taken), continue with the fetched instruction (from PC + 4) 0 stall cycles
     - Else, squash the fetched instruction and re-fetch from the branch target address 1 stall cycle
   - Thus, average branch penalty < 1 cycle

Pipelining

Control Hazard Solutions

1. Static Branch Prediction
2. Delayed Branching

- Design hardware so that control transfer takes place after a few of the following instructions

```
BEQ   R1, R2, target
ADD   R3, R2, R3
```

Pipelining

Recall: Interesting ISA Notes

- For load instructions: the loaded value might not be available in the destination register for use by the instruction immediately following the load
  - LOAD DELAY SLOT
- For control transfer instructions: the transfer of control takes place only following the instruction immediately after the control transfer instruction
  - BRANCH DELAY SLOT

Pipelining

Delayed Branching: Filling Delay Slots

- Instructions that do not affect the branching condition can be put in the delay slot
  - by the compiler
- Where to get instructions to fill delay slots?
  - From the branch target address
    - only valuable when branch is taken
  - From the fall through (branch not taken path)
    - only valuable when branch is not taken
  - From before the branch
    - useful whether branch is taken or not

Pipelining

Delayed Branching…Compiler's Role

- When filled from branch target or fall-through, patch-up code may be needed
- It may still be beneficial, depending on branching frequency
- The more the number of delay slots, the harder it is to fill them usefully

Pipelining

If no instruction can be found…

- The compiler must insert an instruction that does nothing
  - other than occupying the delay slot, being fetched and decoded
  - Example: ADD R0, R0, R0
  - If an instruction that does nothing was included in the instruction set, it would be called a No-Operation instruction, or NOP for short
  - NOP might be included in the assembly language
  - It has practically the same effect as a STALL cycle

Introduction

- Pipelining become universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”
- Beyond this, there are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in PMP processors
  - Compiler-based static approaches
    - Not as successful outside of scientific applications

Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to maximize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls
- Parallelism with basic block is limited
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches

Data Dependence

- Loop-Level Parallelism
  - Unroll loop statically or dynamically
  - Use SIMD (vector processors and GPUs)
- Challenges:
  - Data dependency
    - Instruction *j* is data dependent on instruction *i* if
      - Instruction *i* produces a result that may be used by instruction *j*
      - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i*
- Dependent instructions cannot be executed simultaneously

Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall

- Data dependence conveys:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism

- Dependencies that flow through memory locations are difficult to detect

Name Dependence

- Two instructions use the same name but no flow of information
  - Not a true data dependence, *but is a problem when reordering instructions*
  - *Antidependence*: instruction j writes a register or memory location that instruction i reads
    - Initial ordering (i before j) must be preserved
  - *Output dependence*: instruction i and instruction j write the same register or memory location
    - Ordering must be preserved
- To resolve, use renaming techniques

Other Factors

- Data Hazards
  - Read after write (RAW)
  - Write after write (WAW)
  - Write after read (WAR)
- Control Dependence
  - Ordering of instruction i with respect to a branch instruction
    - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controller by the branch
    - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

Examples

- Example 1:

```
    DADDU R1,R2,R3
    BEQZ R4,L
    DSUBU R1,R1,R6
L:  …
    OR R7,R1,R8
```

- OR instruction dependent on DADDU and DSUBU

- Example 2:

```
    DADDU R1,R2,R3
    BEQZ R12,skip
    DSUBU R4,R5,R6
    DADDU R5,R4,R9
skip:
    OR R7,R8,R9
```

- Assume R4 isn't used after skip
  - Possible to move DSUBU before the branch

Compiler Techniques for Exposing ILP

- Pipeline scheduling
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction
- Example:

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Pipeline Stalls

```
Loop:   L.D  F0,0(R1)
        stall
        ADD.D F4,F0,F2
        stall
        stall
        S.D F4,0(R1)
        DADDUI R1,R1,#-8
        stall (assume integer load latency is 1)
        BNE R1,R2,Loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Pipeline Scheduling

Scheduled code:

```
Loop:   L.D  F0,0(R1)
        DADDUI R1,R1,#-8
        ADD.D F4,F0,F2
        stall
        stall
        S.D F4,8(R1)
        BNE R1,R2,Loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Loop Unrolling

- Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program.
- basically remove or reduce iterations.
- Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

Loop Unrolling


```
// This program does not uses loop unrolling.
#include<stdio.h>

int main(void)
{
 for (int i=0; i<5; i++)
       printf("Hello\n");
//print hello 5 times

 return 0;
}
```

```
// This program uses loop unrolling.
#include<stdio.h>

int main(void)
{
  // unrolled the for loop in program 1
  printf("Hello\n");
  printf("Hello\n");
  printf("Hello\n");
  printf("Hello\n");
  printf("Hello\n");

  return 0;
}
```

Loop Unrolling

- **Advantages:**
- Increases program efficiency.
- Reduces loop overhead.
- If statements in loop are not dependent on each other, they can be executed in parallel.
- **Disadvantages:**
- Increased program code size, which can be undesirable.
- Possible increased usage of register in a single iteration to store temporary variables which may reduce performance.

Loop Unrolling

- Loop unrolling
  - Unroll by a factor of 4 (assume # elements is divisible by 4)
  - Eliminate unnecessary instructions

```
Loop:   L.D F0,0(R1)
        ADD.D F4,F0,F2
        S.D F4,0(R1) ;drop DADDUI & BNE
        L.D F6,-8(R1)
        ADD.D F8,F6,F2
        S.D F8,-8(R1) ;drop DADDUI & BNE
        L.D F10,-16(R1)
        ADD.D F12,F10,F2
        S.D F12,-16(R1) ;drop DADDUI & BNE
        L.D F14,-24(R1)
        ADD.D F16,F14,F2
        S.D F16,-24(R1)
        DADDUI R1,R1,#-32
        BNE R1,R2,Loop
```

- note: number of live registers vs. original loop

Loop Unrolling/Pipeline Scheduling

- Pipeline schedule the unrolled loop:

```
Loop:   L.D F0,0(R1)
        L.D F6,-8(R1)
        L.D F10,-16(R1)
        L.D F14,-24(R1)
        ADD.D F4,F0,F2
        ADD.D F8,F6,F2
        ADD.D F12,F10,F2
        ADD.D F16,F14,F2
        S.D F4,0(R1)
        S.D F8,-8(R1)
        DADDUI R1,R1,#-32
        S.D F12,16(R1)
        S.D F16,8(R1)
        BNE R1,R2,Loop
```

Strip Mining

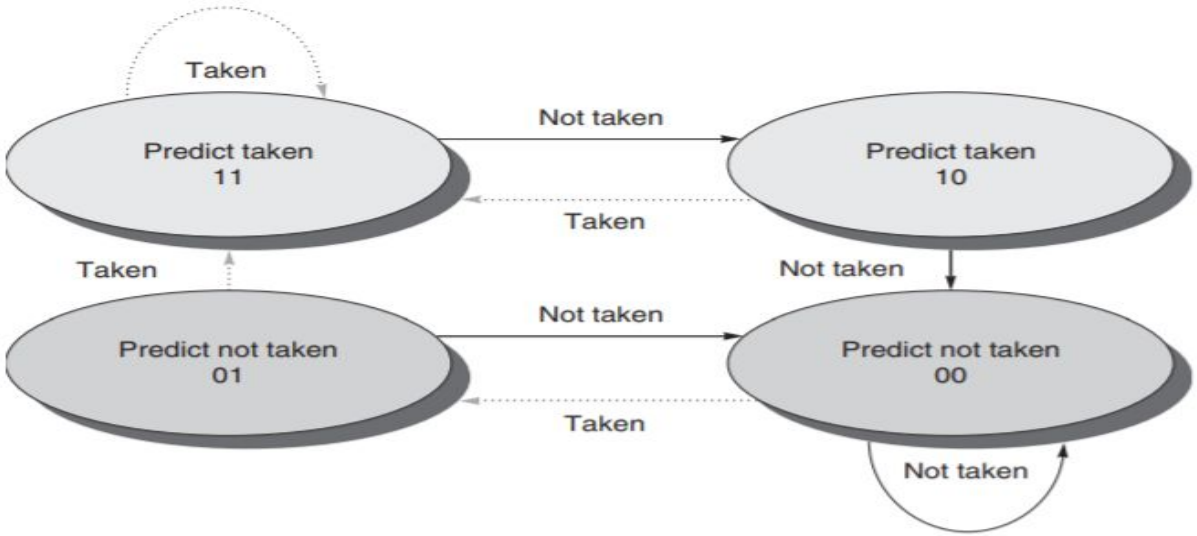
- Strip-mining, also known as loop sectioning, is a loop transformation technique for enabling SIMD-encodings of loops, as well as a means of improving memory performance.
- By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure in two ways:
- It increases the temporal and spatial locality in the data cache if the data are reusable in different passes of an algorithm.
- It reduces the number of iterations of the loop by a factor of the length of each vector, or number of operations being performed per SIMD operation.

Strip Mining

- Unknown number of loop iterations?
  - Number of iterations = *n*
  - Goal: make *k* copies of the loop body
  - Generate pair of loops:
    - First executes *n* mod *k* times
    - Second executes *n* / *k* times
    - "Strip mining"

Branch Prediction

- Basic 2-bit predictor:
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction



Branch Prediction

- Correlating predictor:
  - The 2-bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch.
  - It may be possible to improve the prediction accuracy if we also look at the recent behavior of other branches rather than just the branch we are trying to predict.
  - if (aa==2) aa=0;
  - if (bb==2) bb=0;
  - if (aa!=bb) {

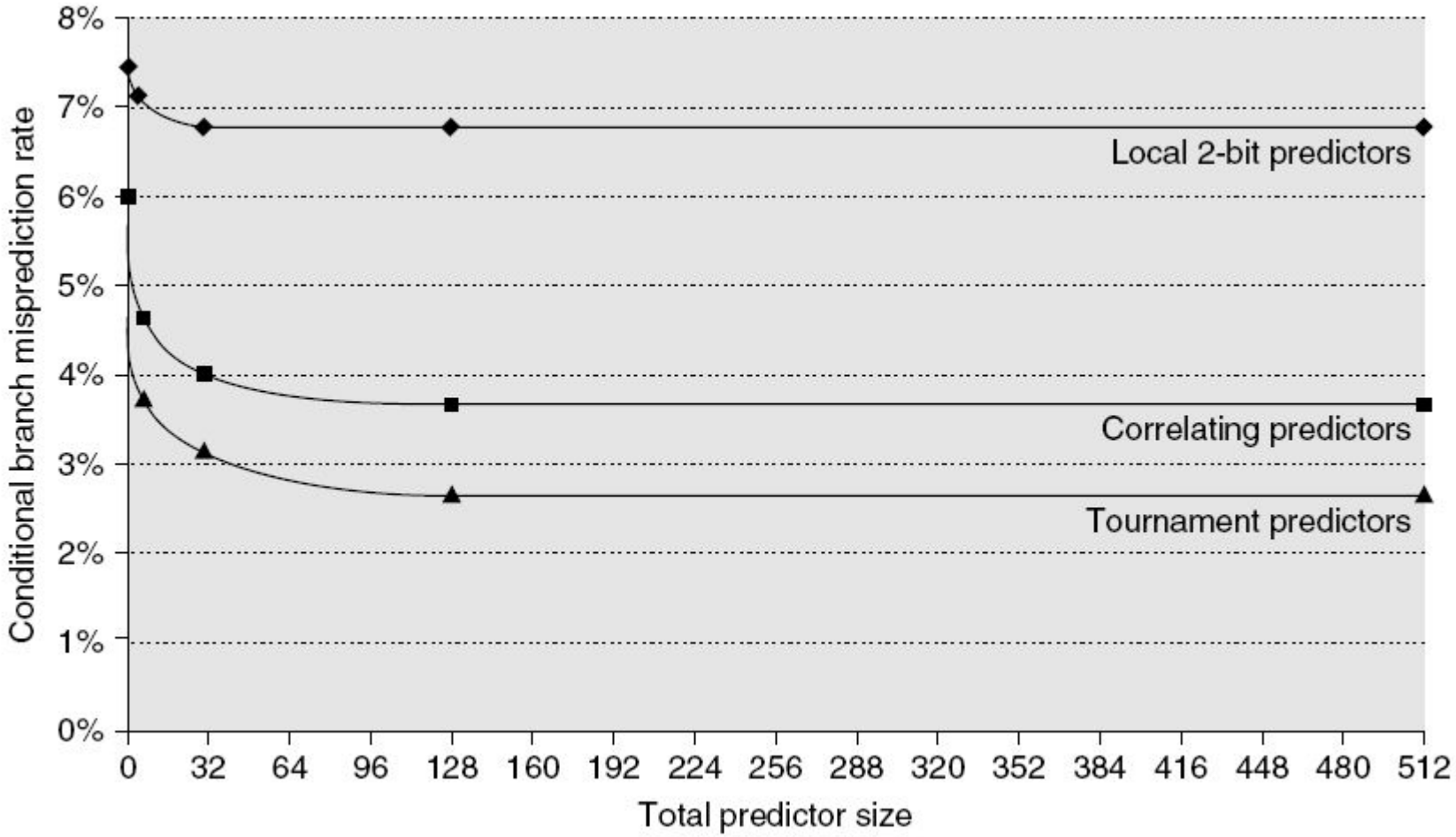
Branch Prediction

- Correlating predictor:
  - Let's label these branches b1, b2, and b3.
  - The key observation is that the behavior of branch b3 is correlated with the behavior of branches b1 and b2.
  - Clearly, if branches b1 and b2 are both not taken (i.e., if the conditions both evaluate to true and aa and bb are both assigned 0), then b3 will be taken, since aa and bb are clearly equal.
  - A predictor that uses only the behavior of a single branch to predict the outcome of that branch can never capture this behavior.
  - Branch predictors that use the behavior of other branches to make a prediction are called correlating predictors or two-level predictors

Branch Prediction

- Tournament predictor:
  - Combine correlating predictor with local predictor

Branch Prediction Performance



Branch predictor performance

Dynamic Scheduling

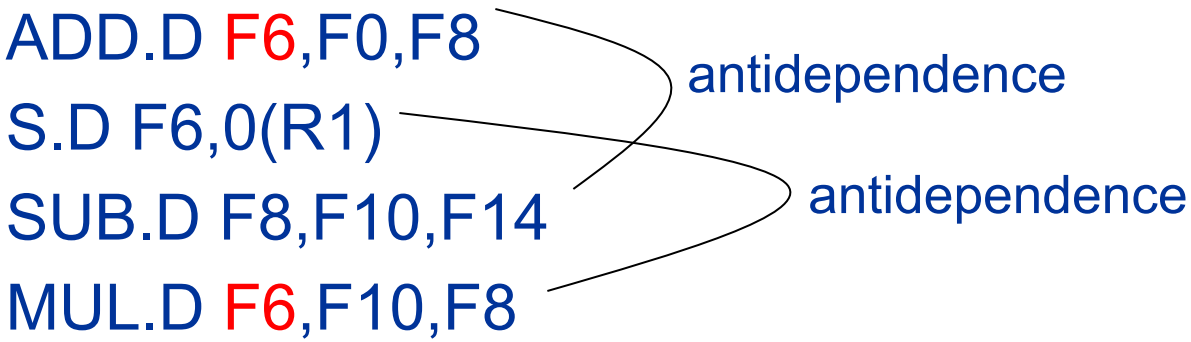
- Rearrange order of instructions to reduce stalls while maintaining data flow
- Run time, H/W
- Advantages:
  - Compiler doesn't need to have knowledge of microarchitecture
  - Handles cases where dependencies are unknown at compile time
- Disadvantage:
  - Substantial increase in hardware complexity
  - Complicates exceptions

Dynamic Scheduling

- Dynamic scheduling implies:
  - Out-of-order execution
  - Out-of-order completion
- Creates the possibility for WAR and WAW hazards
- Tomasulo's Approach
  - Tracks when operands are available
  - Introduces register renaming in hardware
    - Minimizes WAW and WAR hazards

Register Renaming

- Example:



+ name dependence with F6

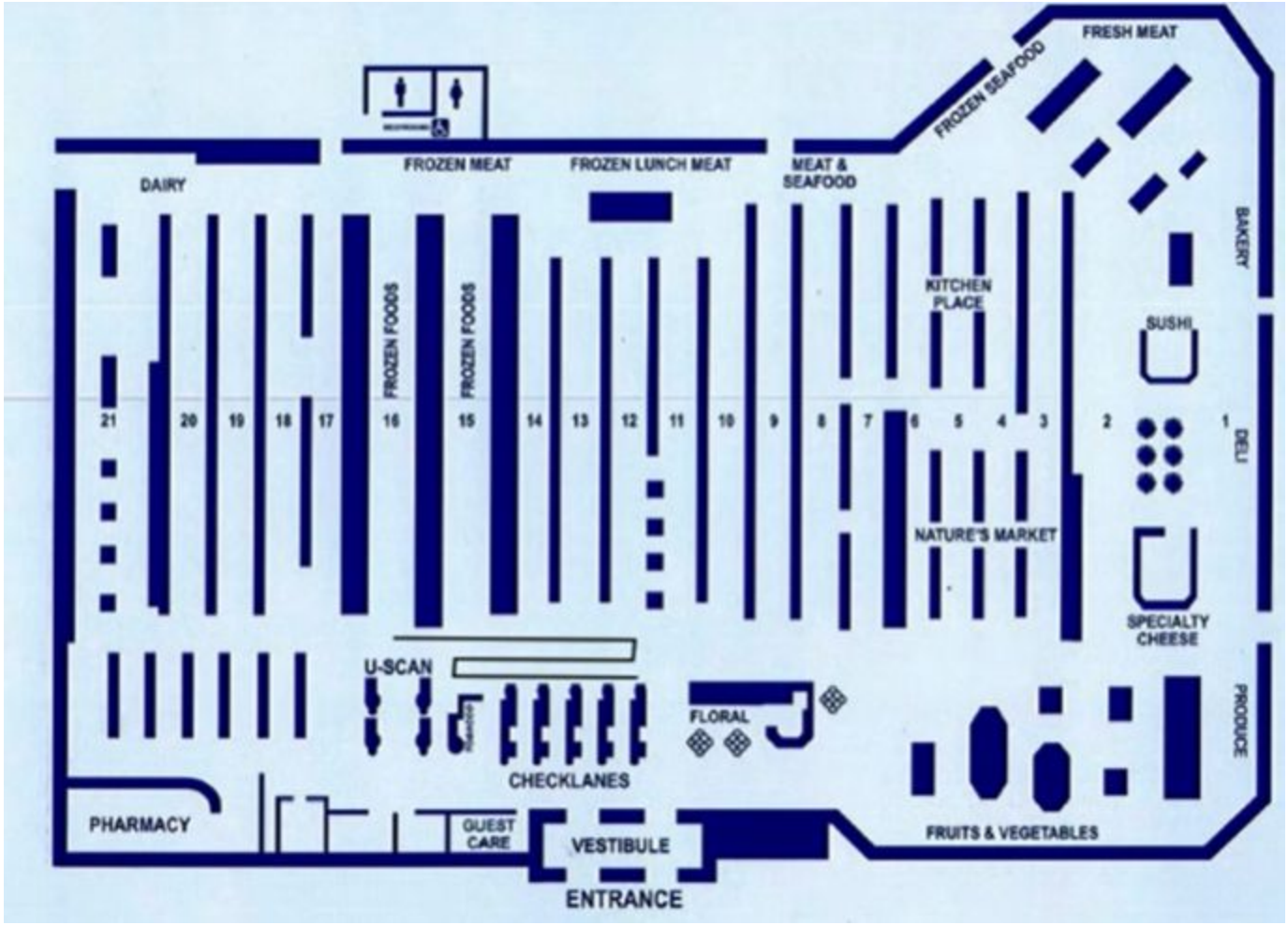
Register Renaming

- Example:

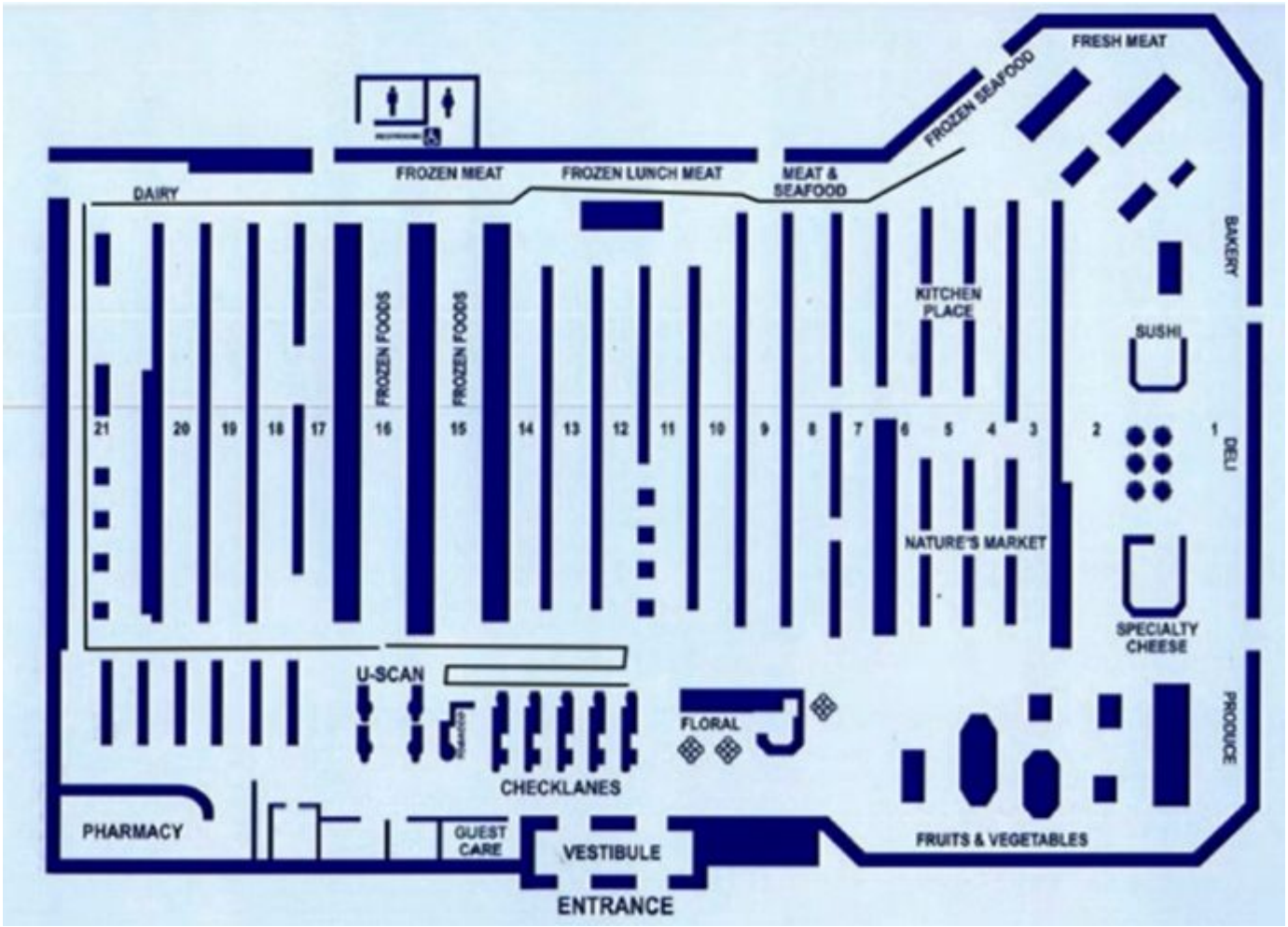
```
DIV.D F0,F2,F4
ADD.D S,F0,F8
S.D S,0(R1)
SUB.D T,F10,F14
MUL.D F6,F10,T
```

- Now only RAW hazards remain, which can be strictly ordered

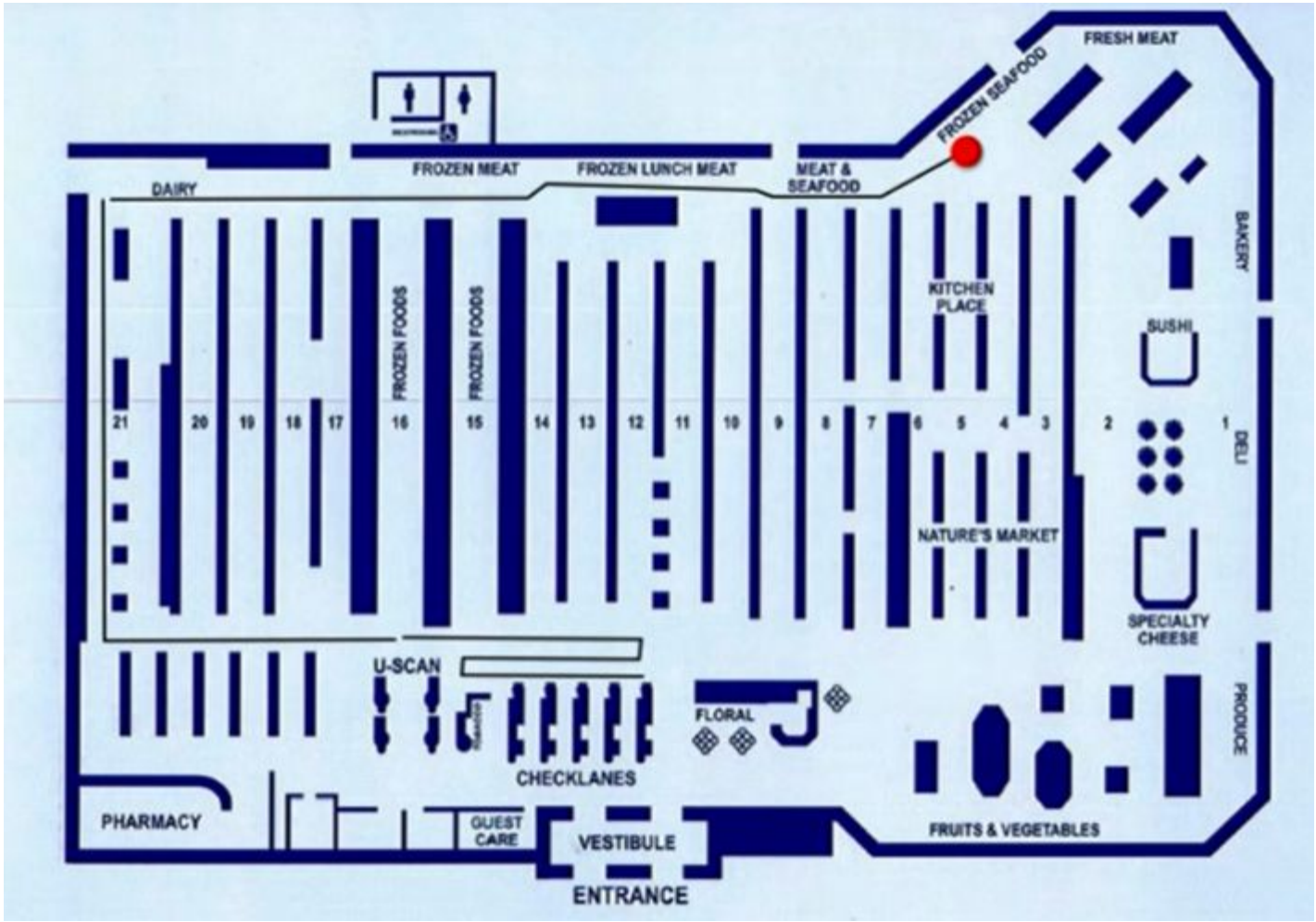
Algorithm



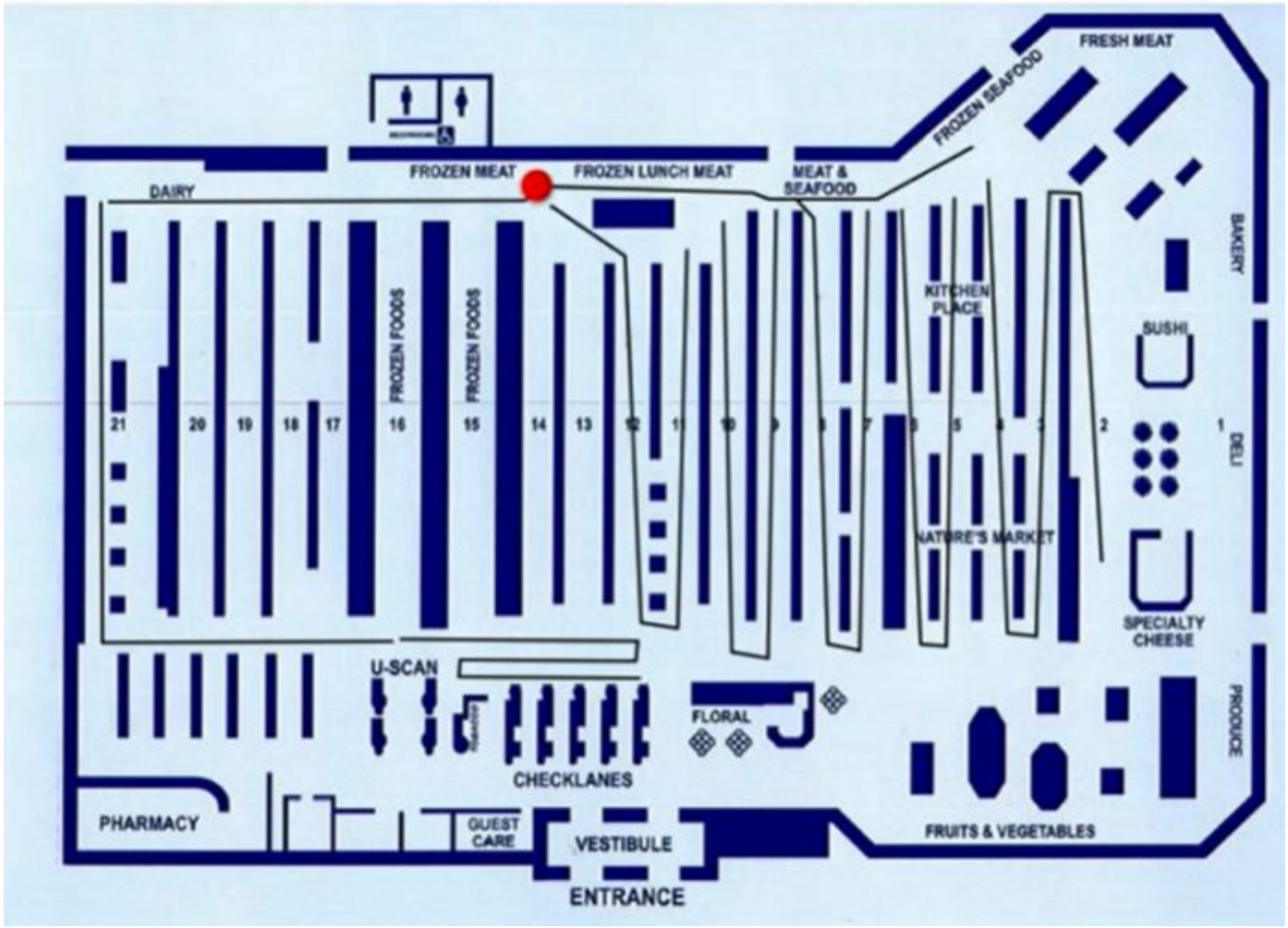
Algorithm



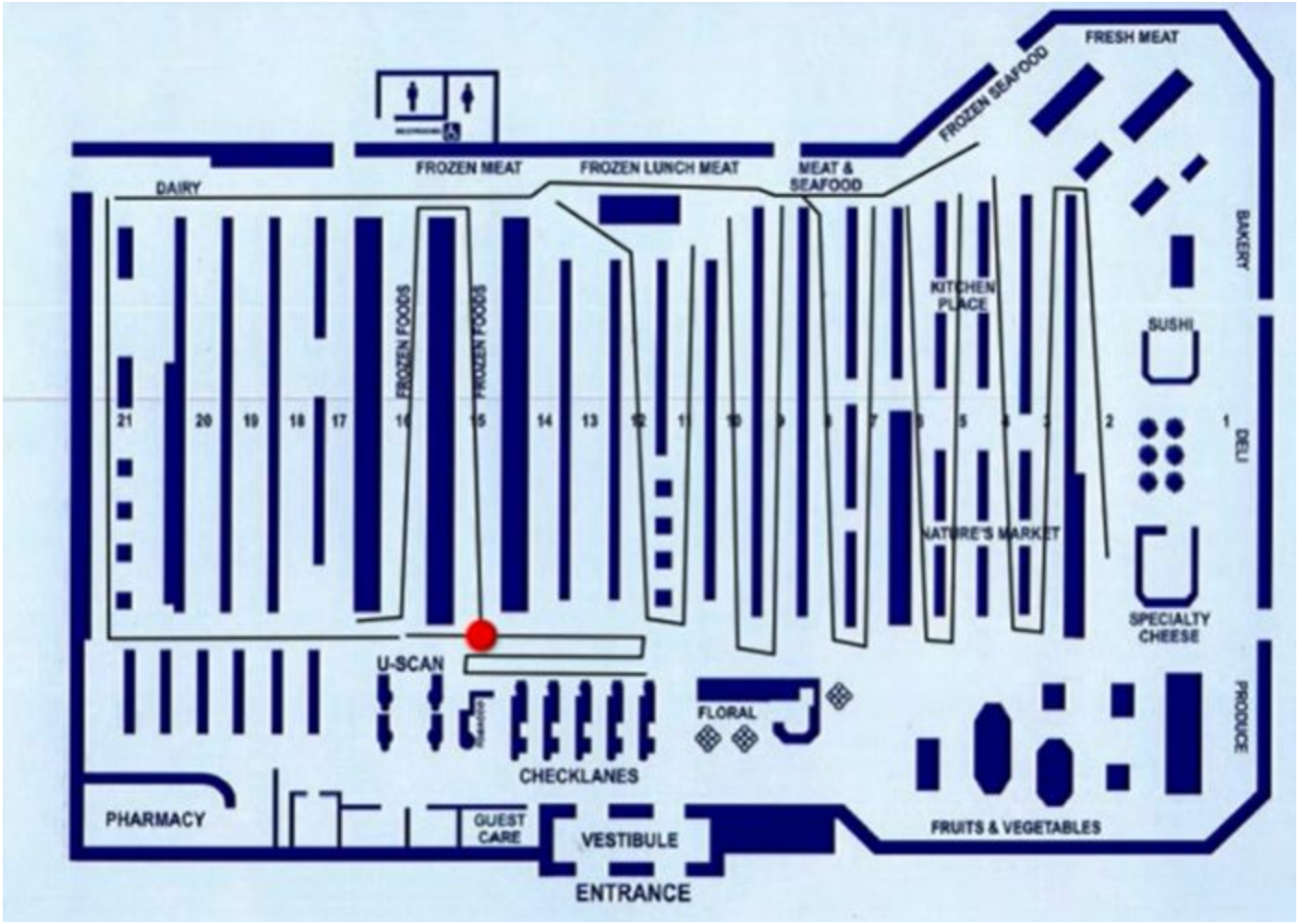
Algorithm



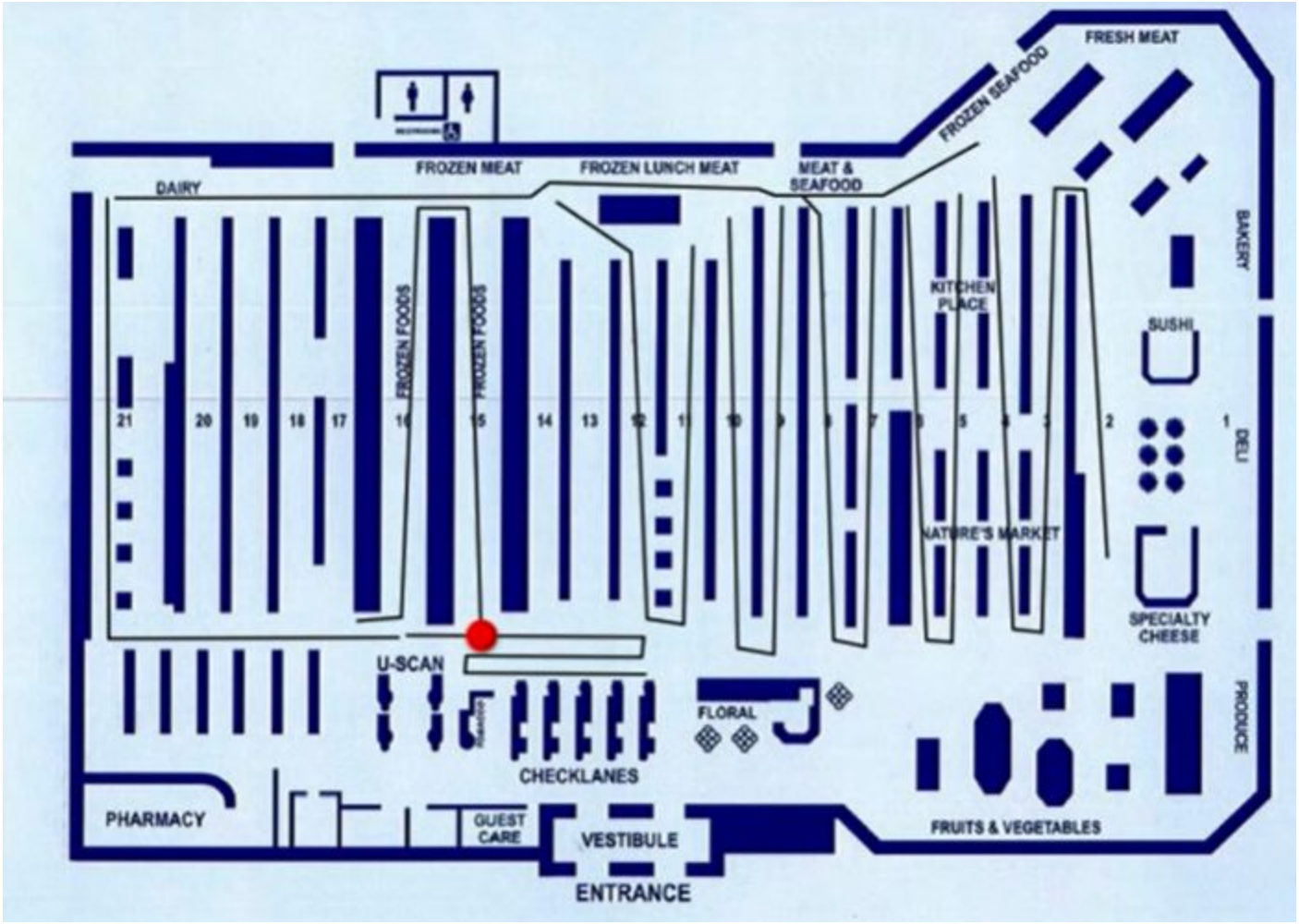
Algorithm



Algorithm

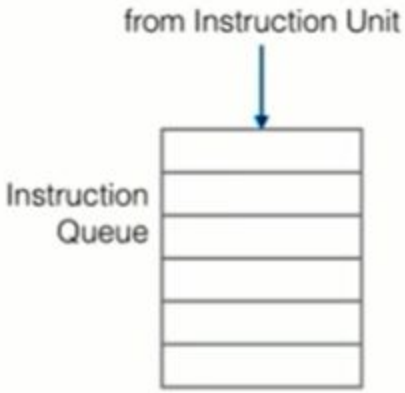


Algorithm



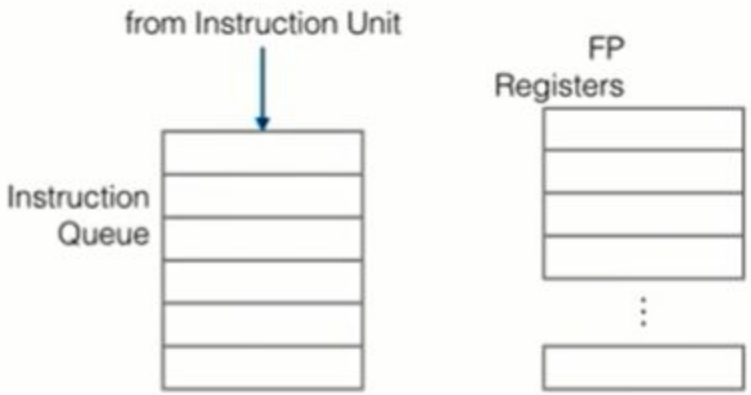
Algorithm

Part 1
The Architecture

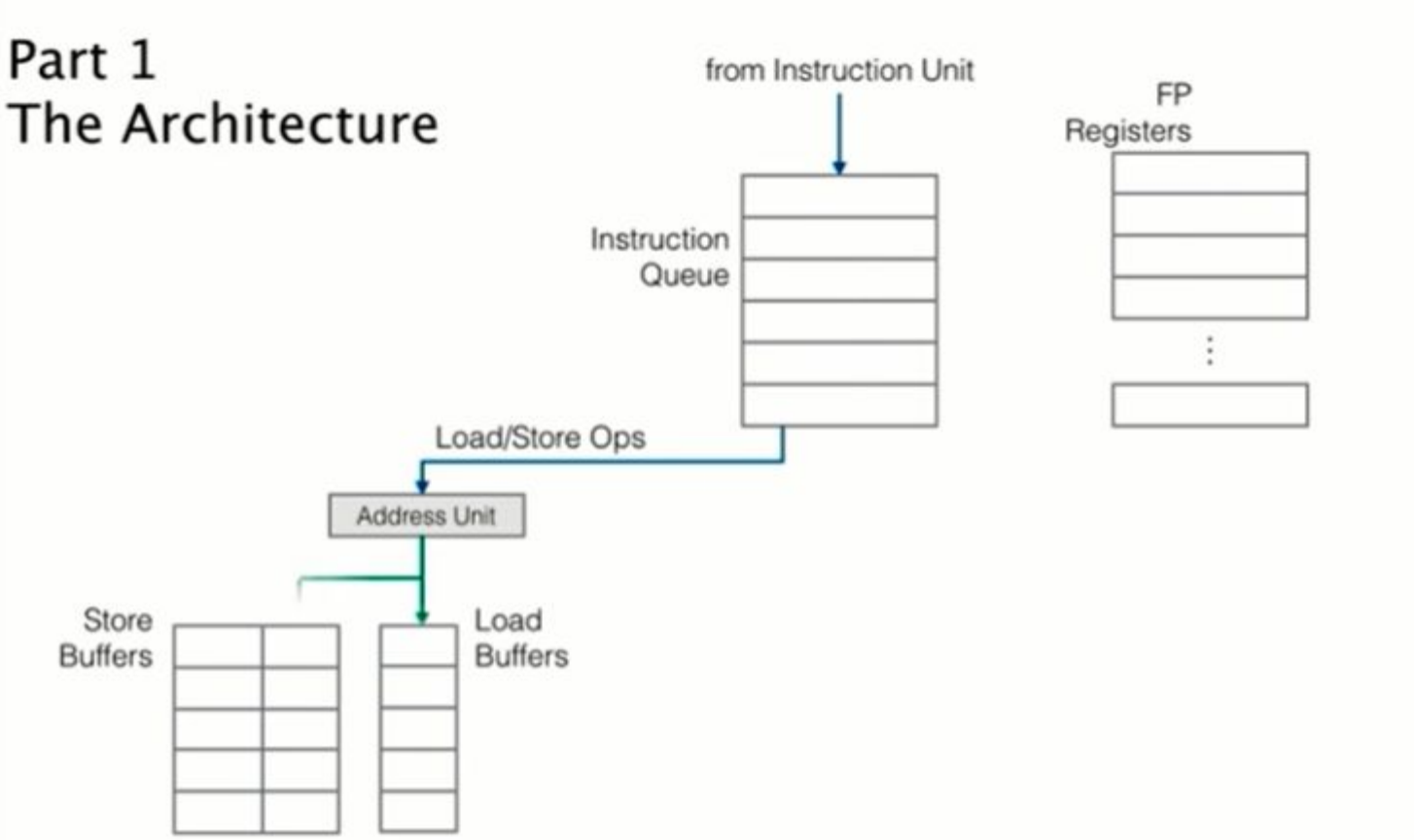


Dynamic Scheduling: Tomasulo's Algorithm

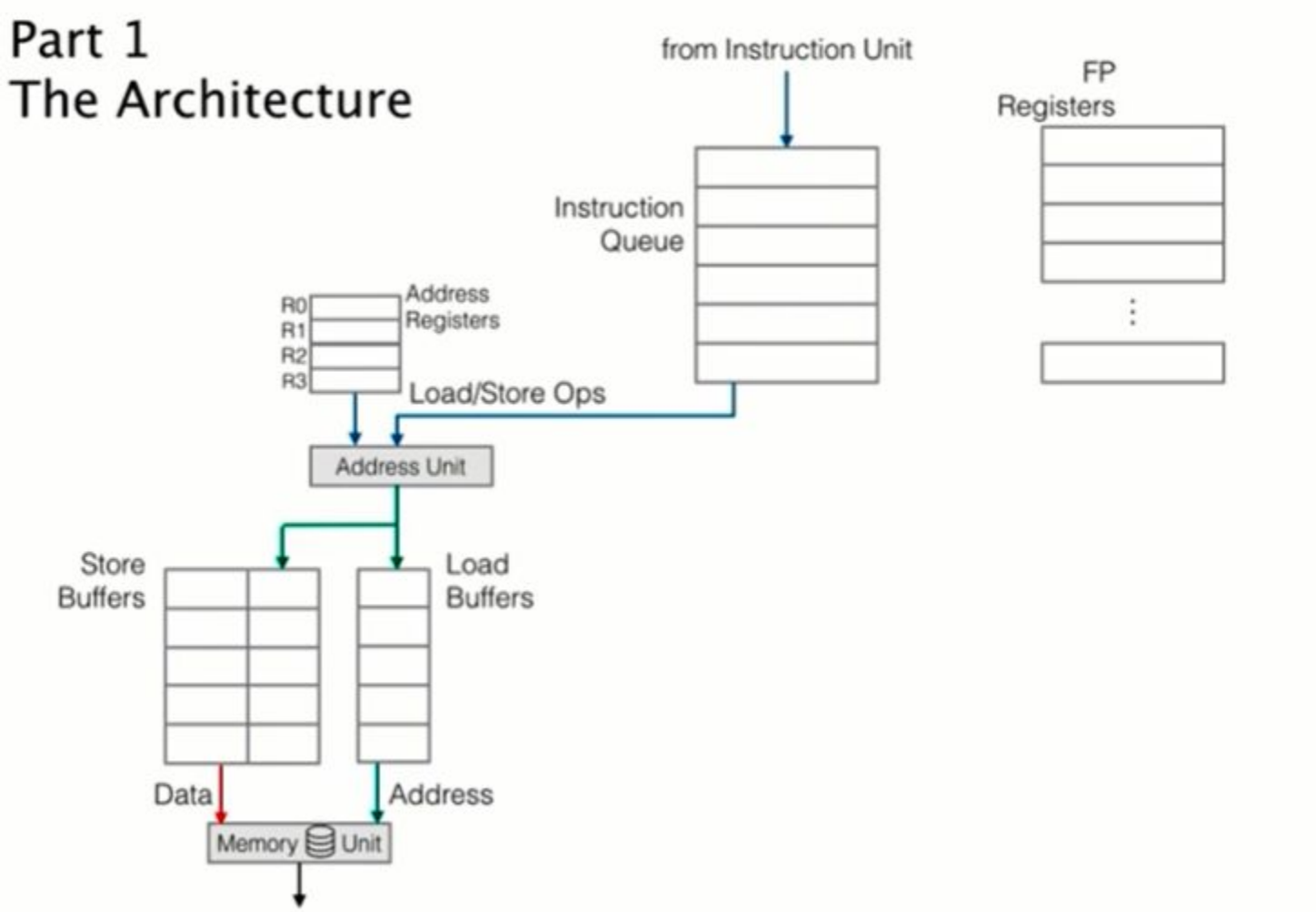
Part 1
The Architecture



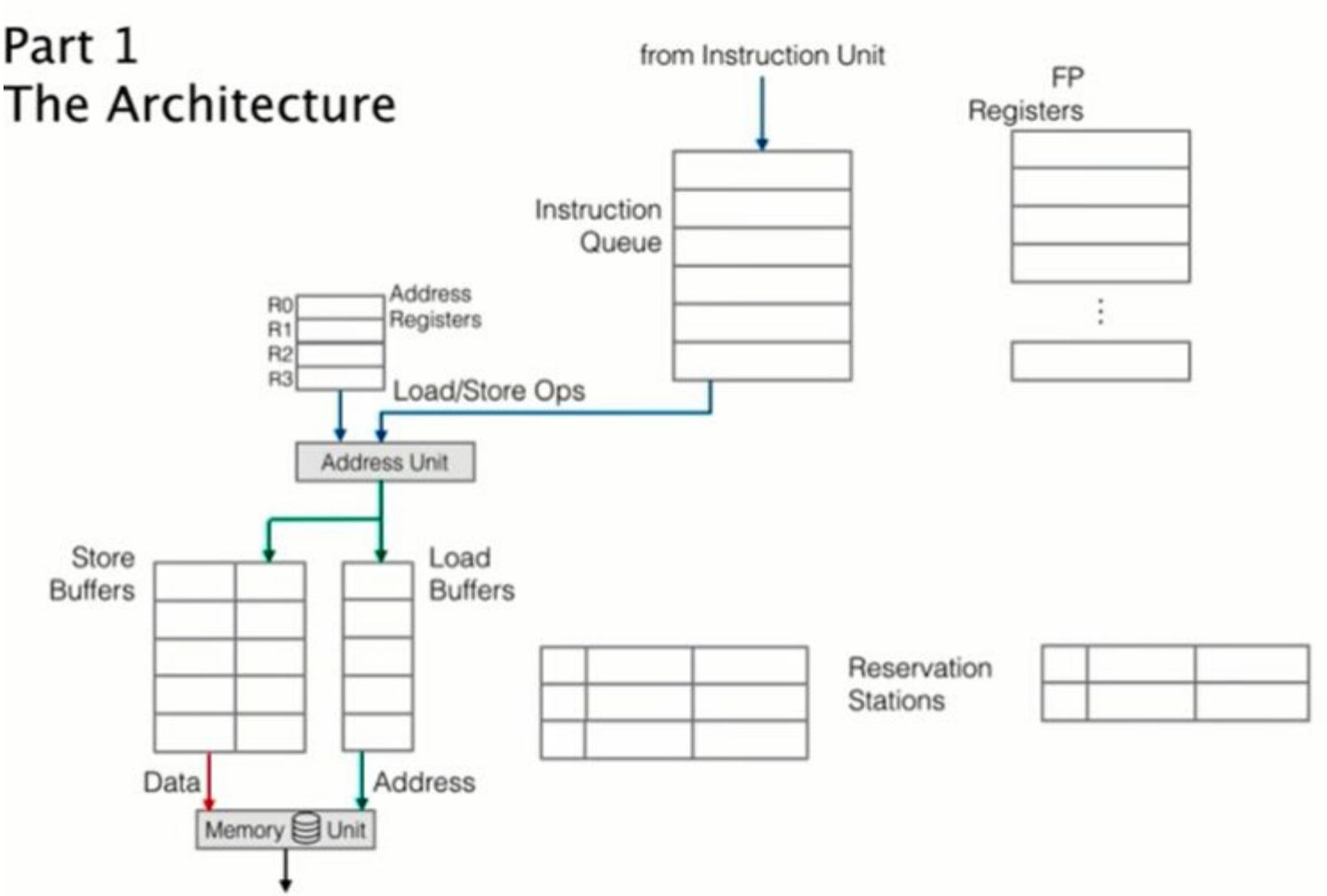
Dynamic Scheduling: Tomasulo's Algorithm



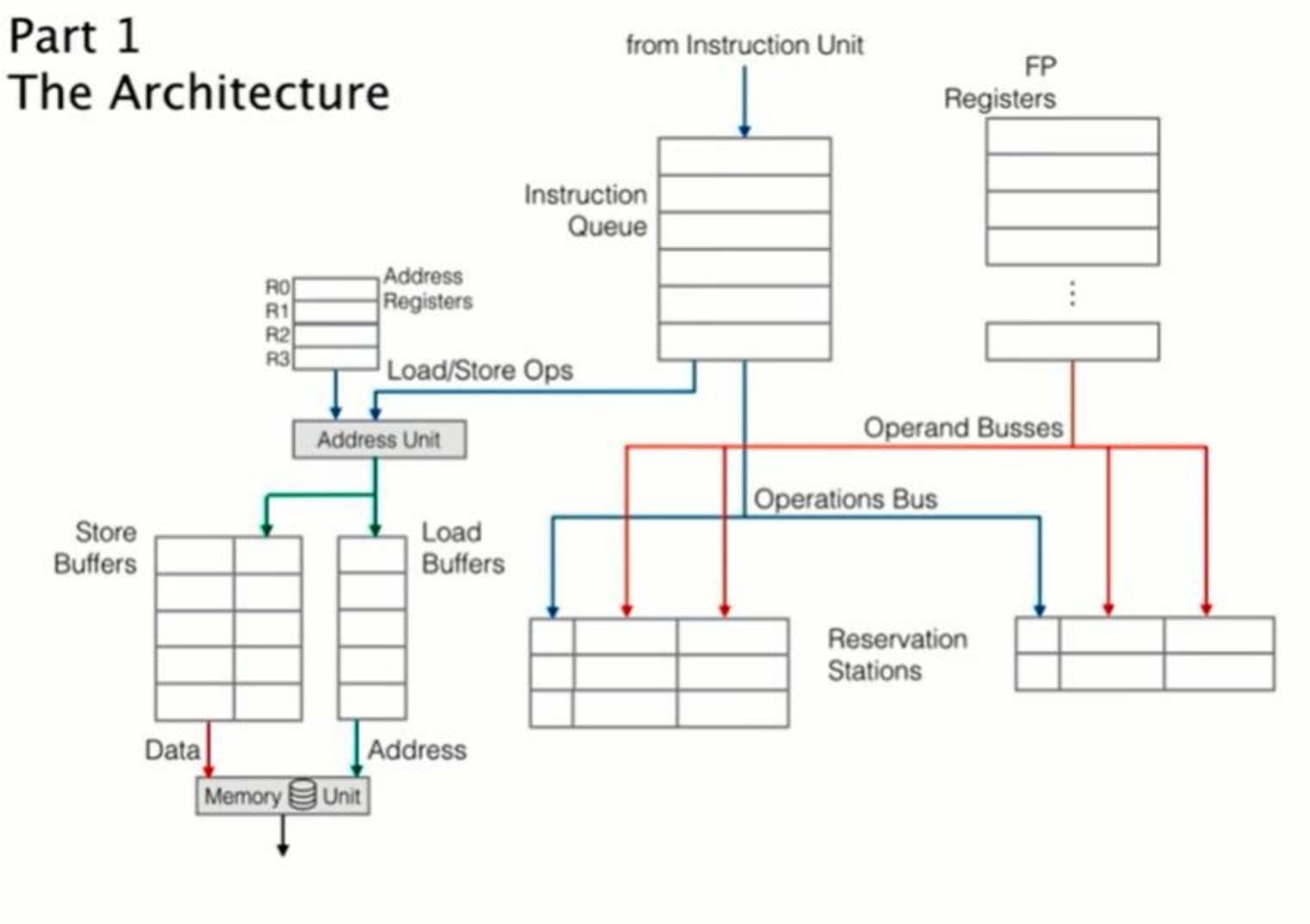
Algorithm



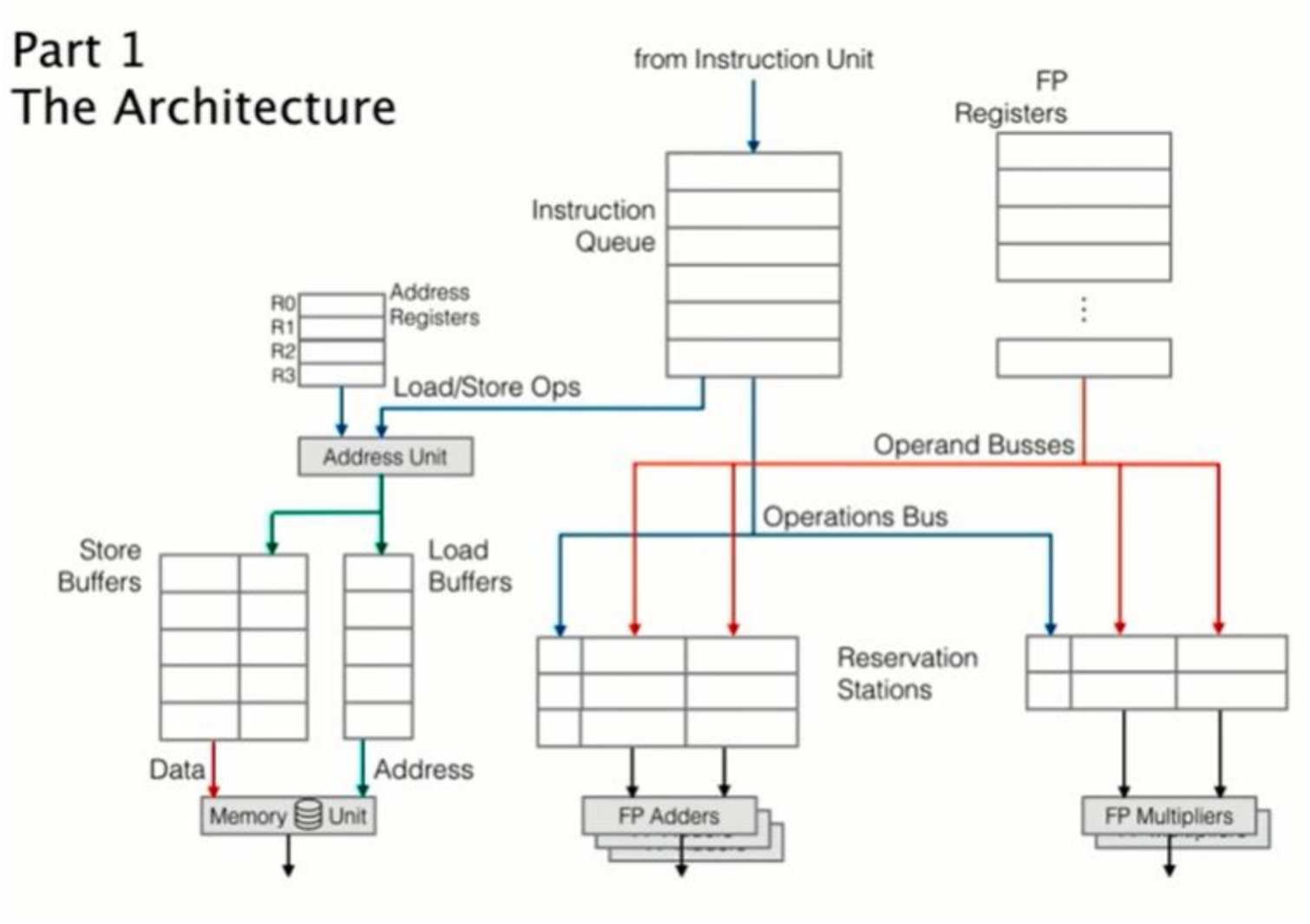
Algorithm



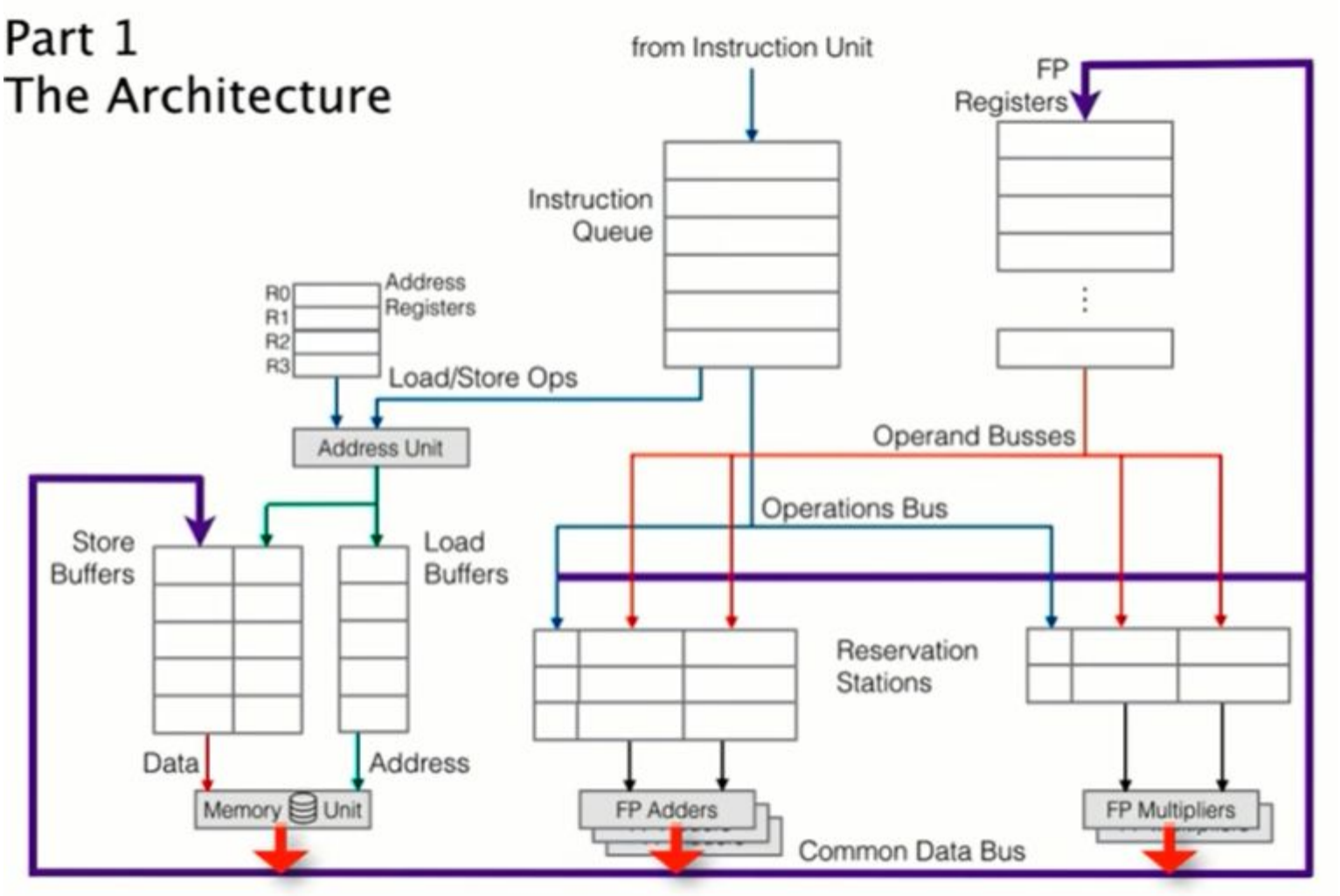
Dynamic Scheduling: Tomasulo's Algorithm



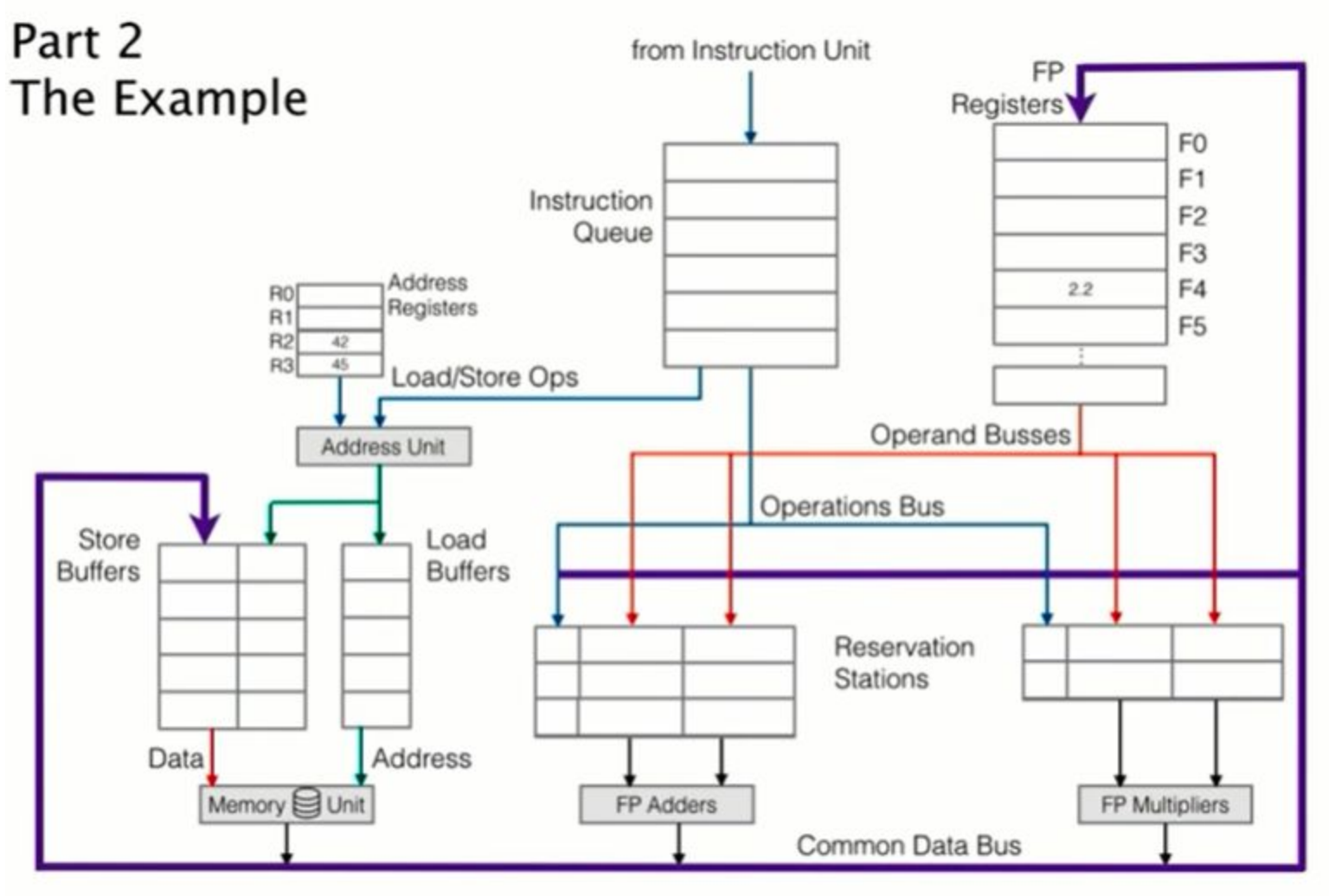
Dynamic Scheduling: Tomasulo's Algorithm



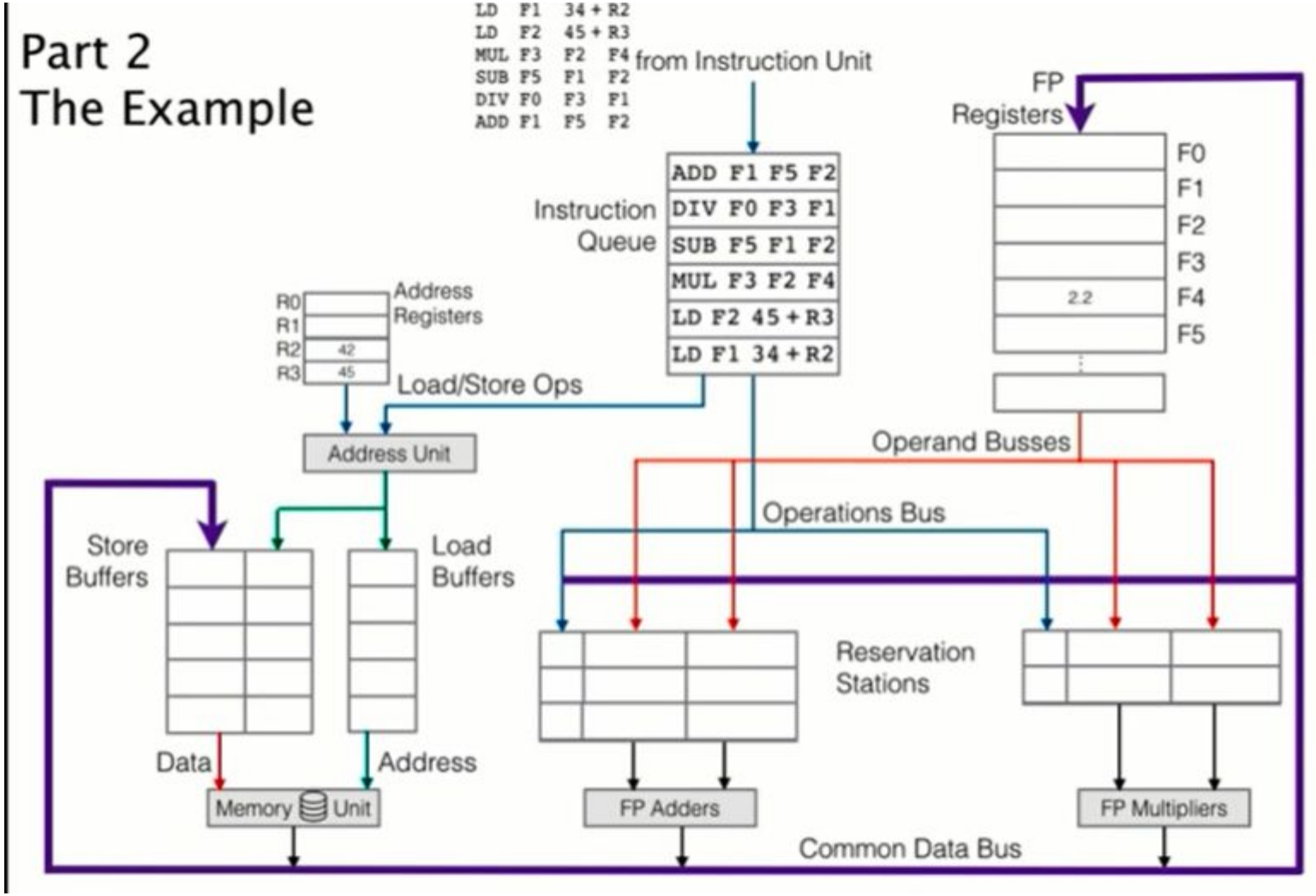
Dynamic Scheduling: Tomasulo's Algorithm



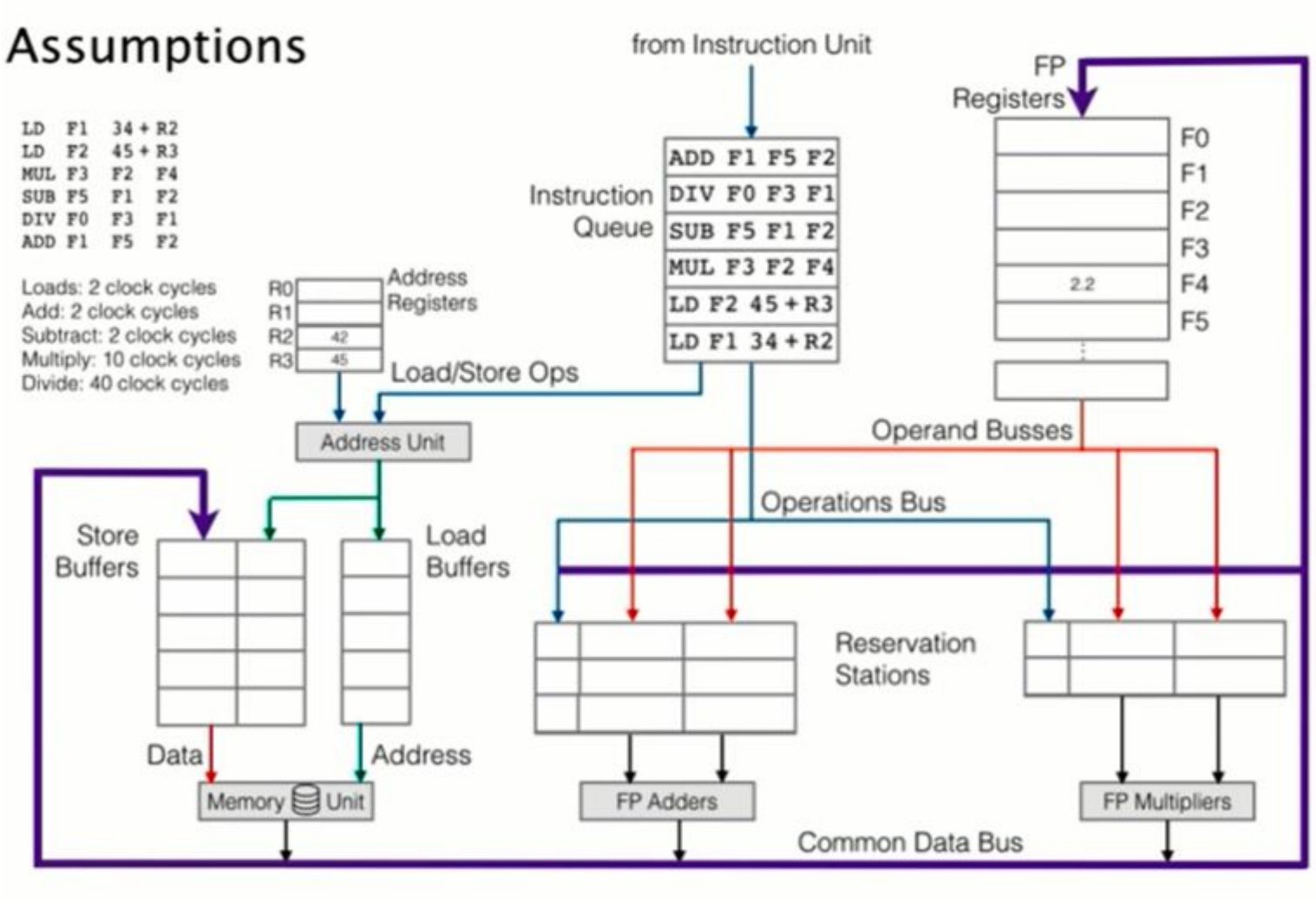
Dynamic Scheduling: Tomasulo's Algorithm



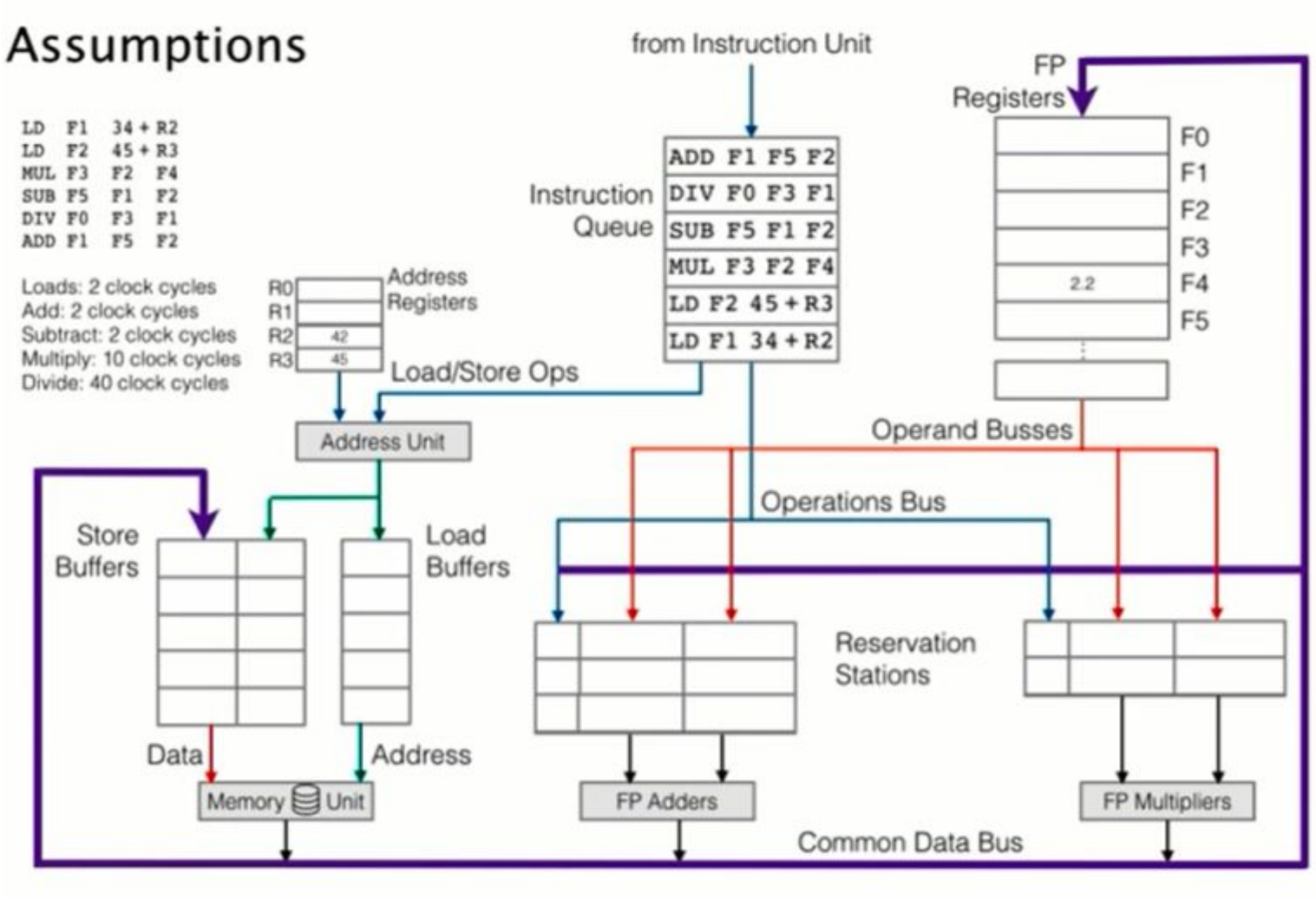
Algorithm



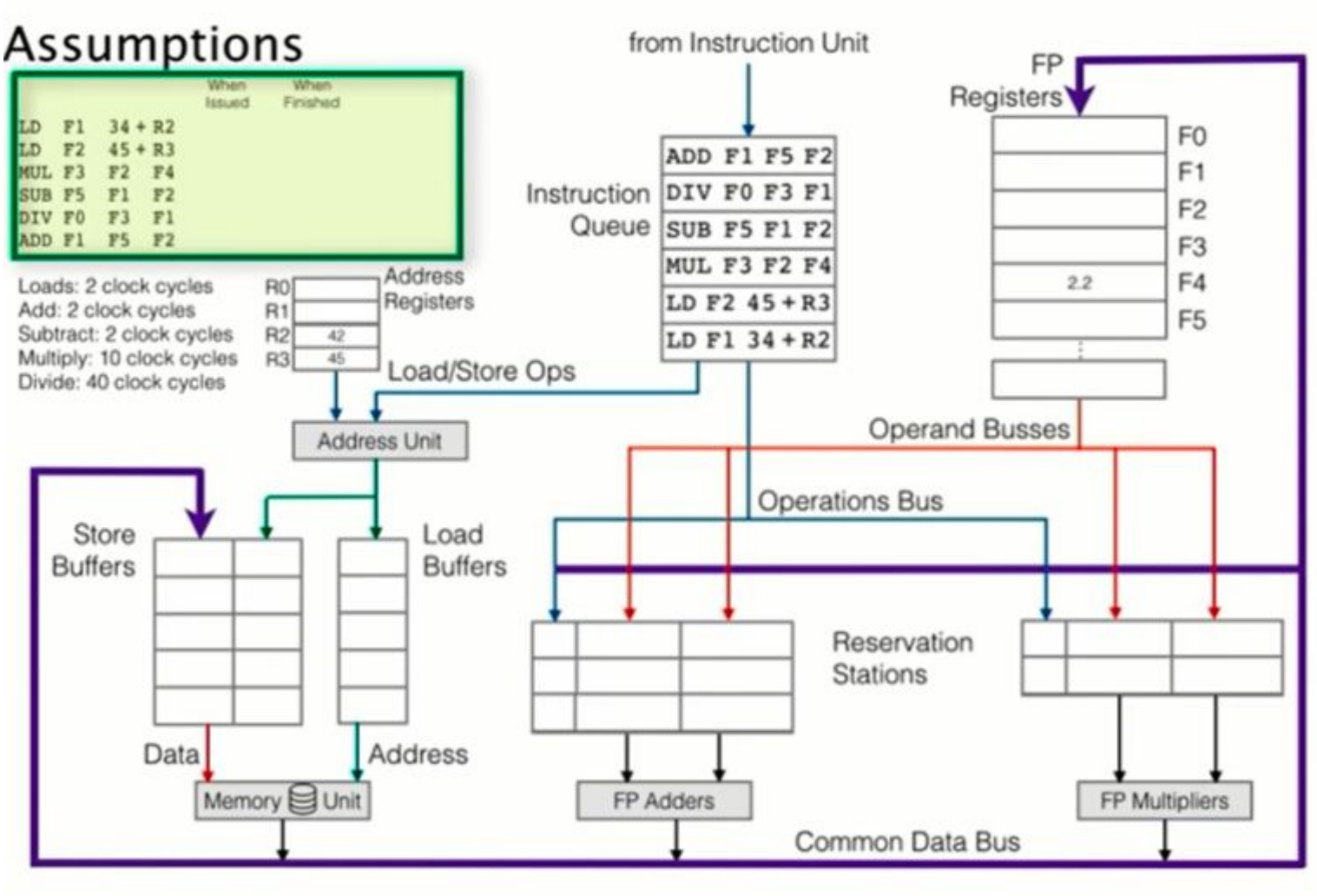
Dynamic Scheduling: Tomasulo's Algorithm



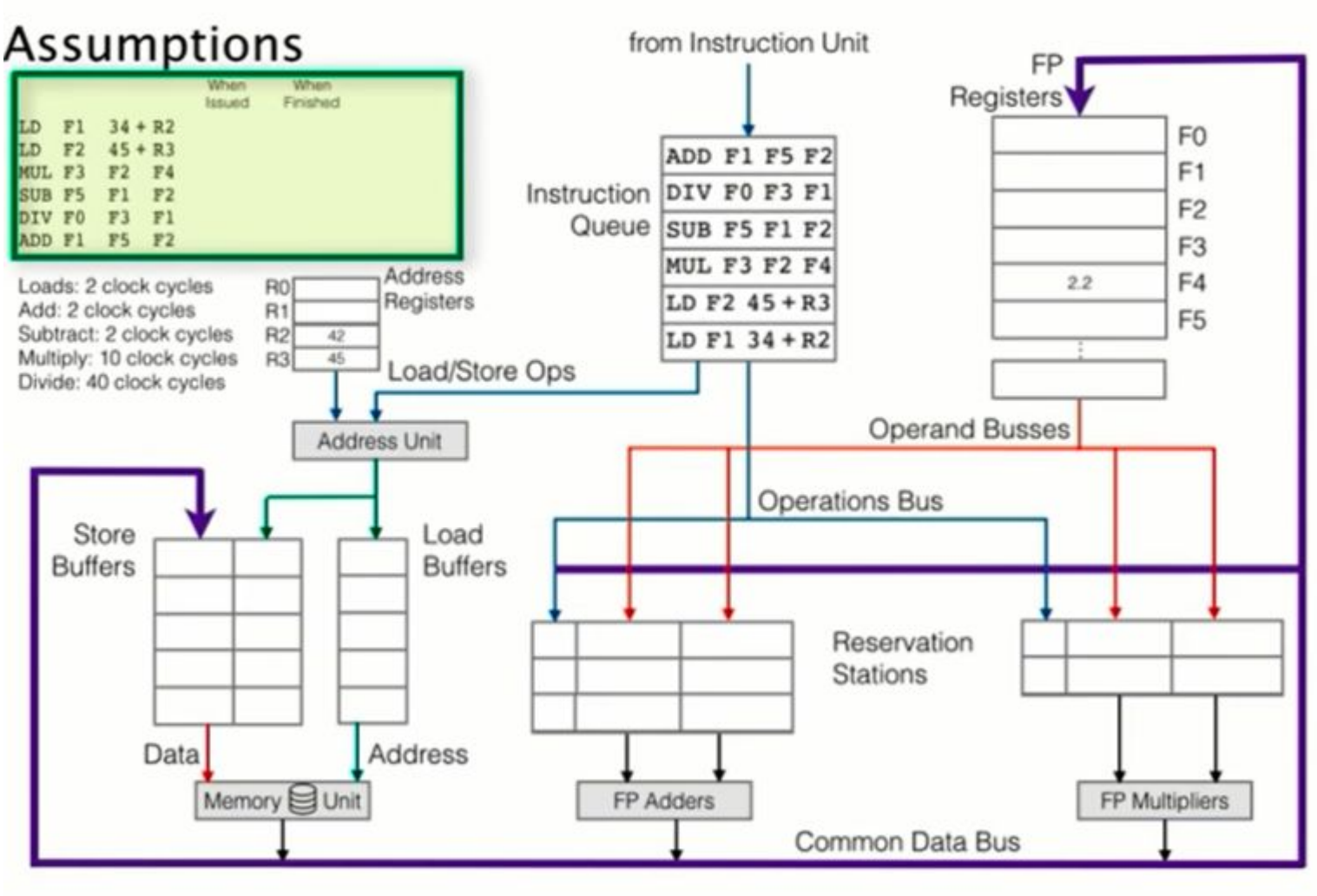
Dynamic Scheduling: Tomasulo's Algorithm



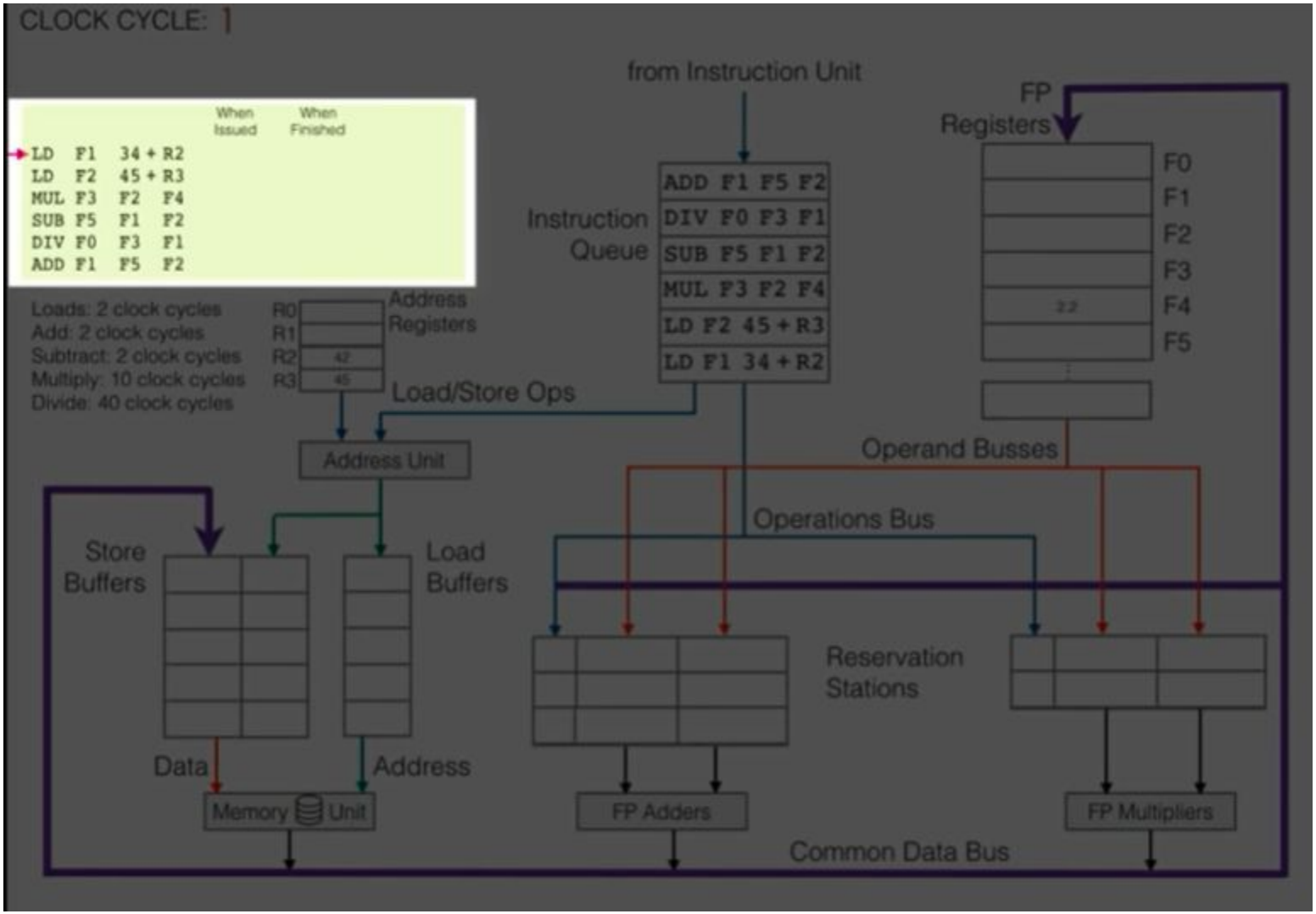
Algorithm



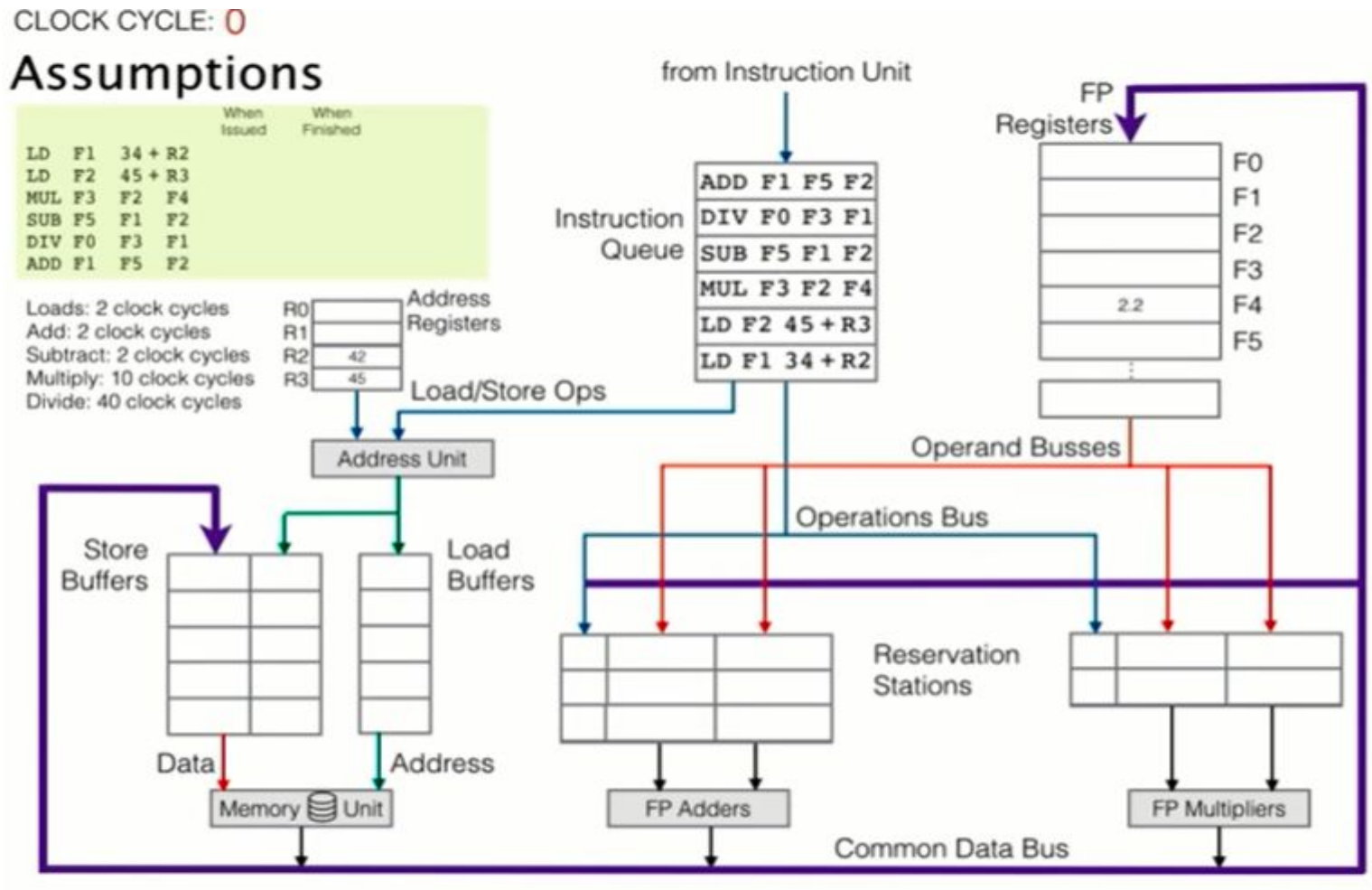
Algorithm



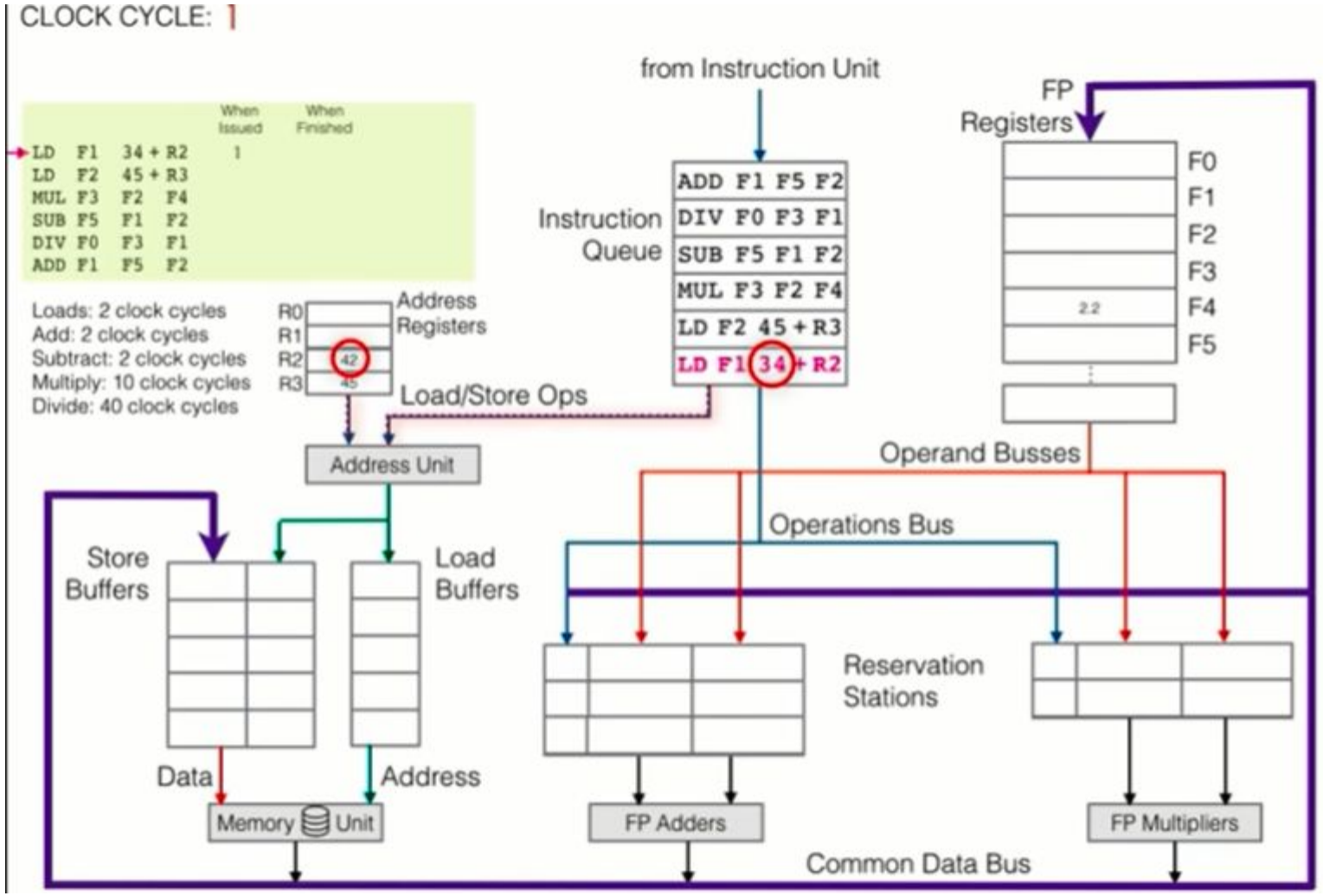
Algorithm



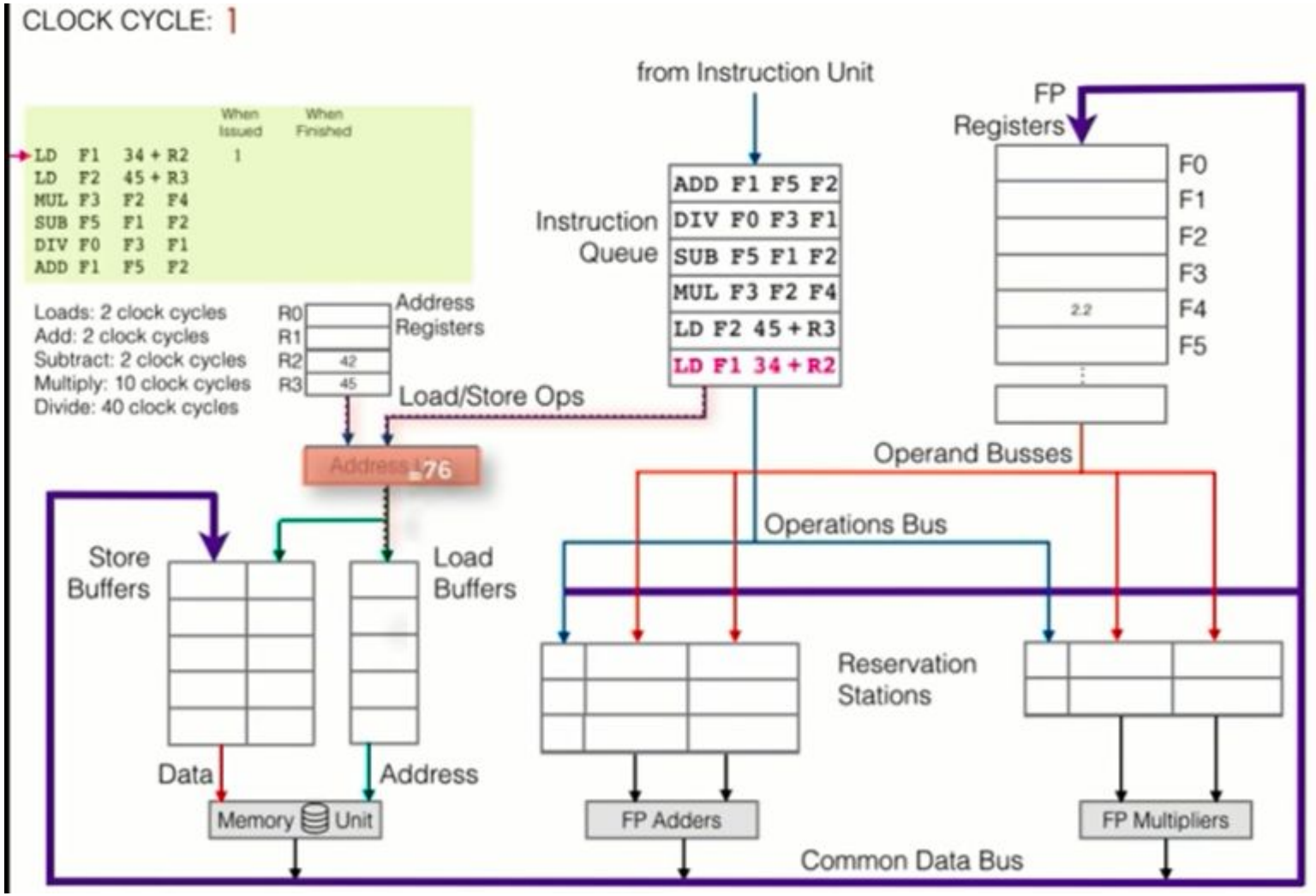
Algorithm



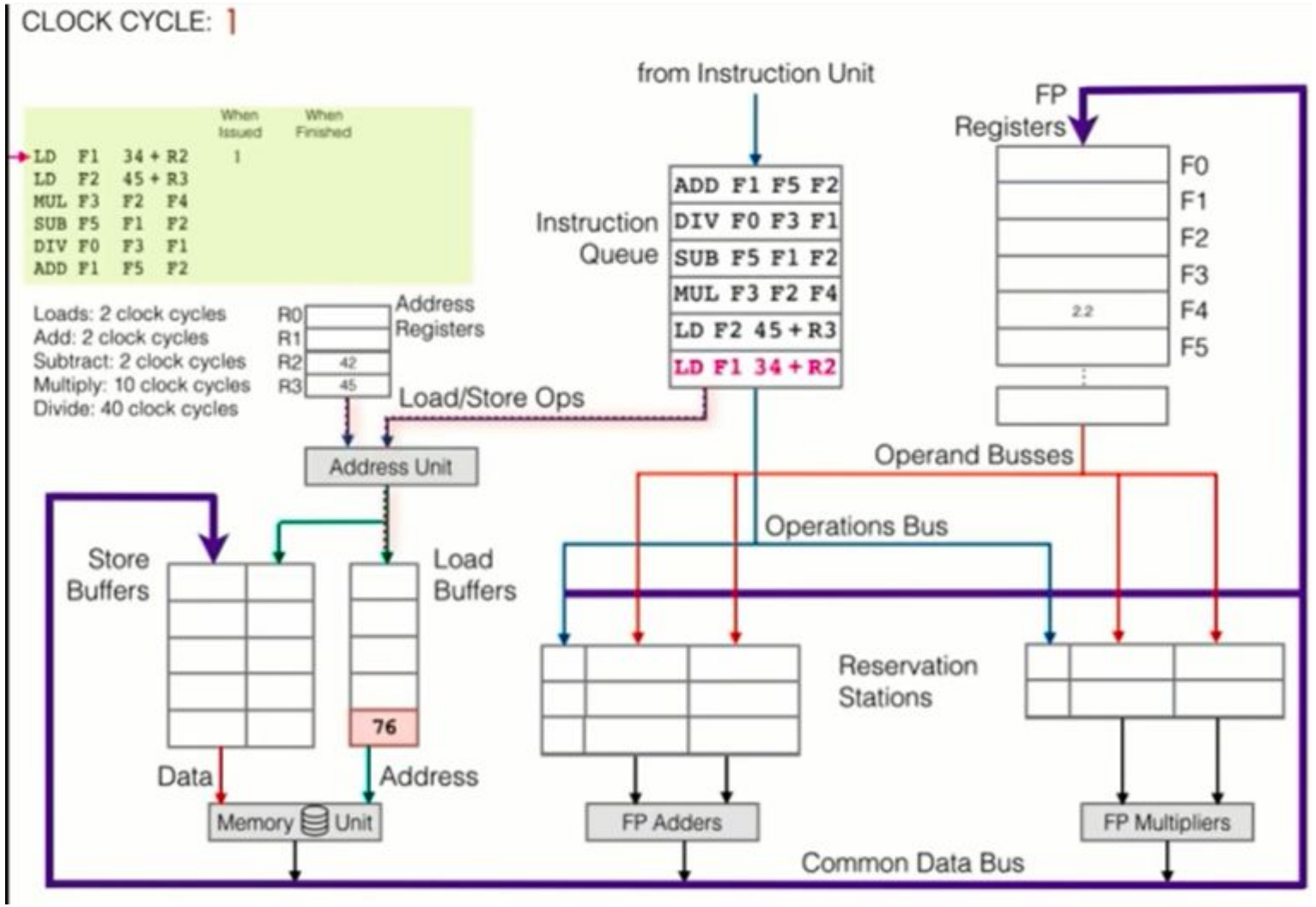
Algorithm



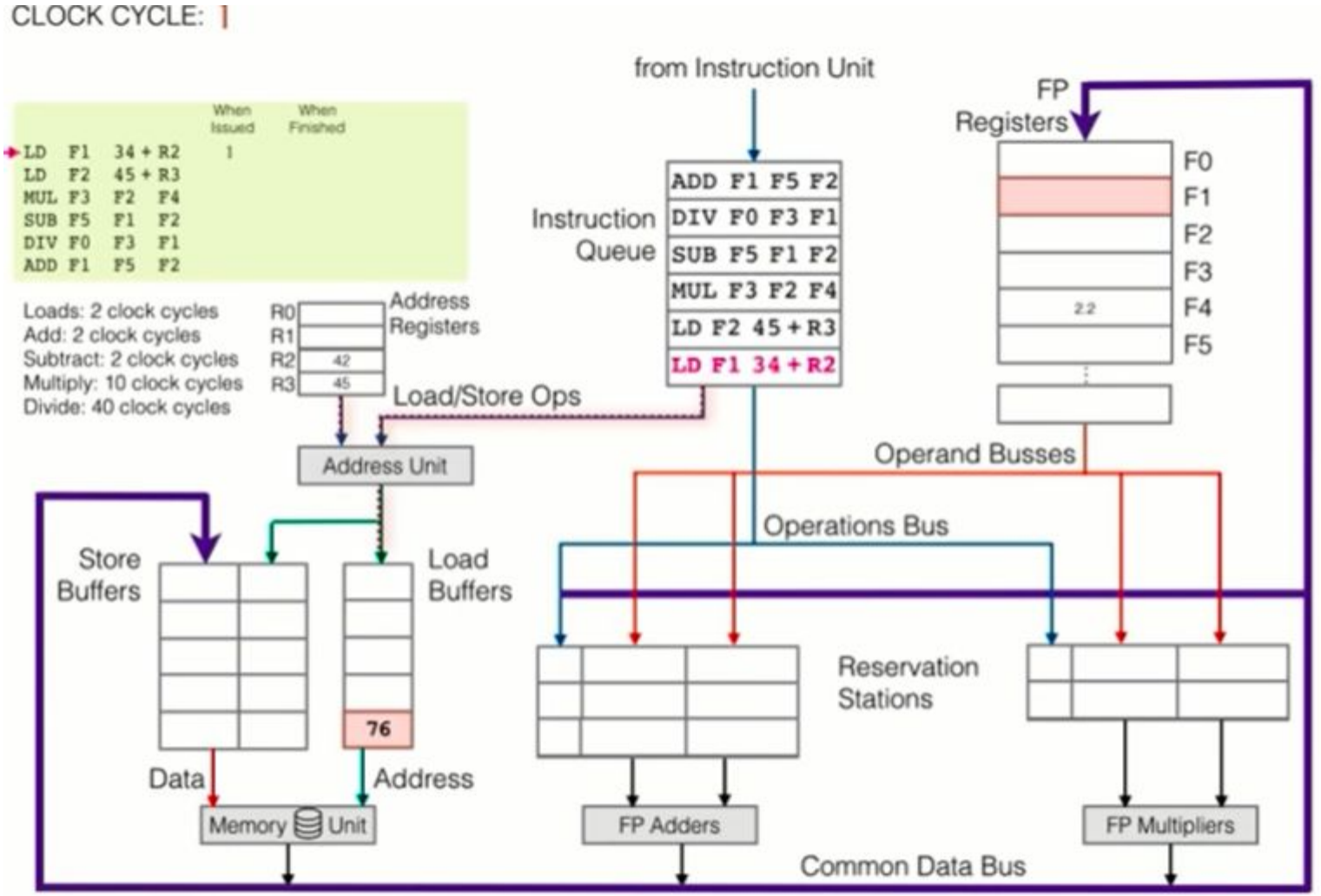
Dynamic Scheduling: Tomasulo's Algorithm



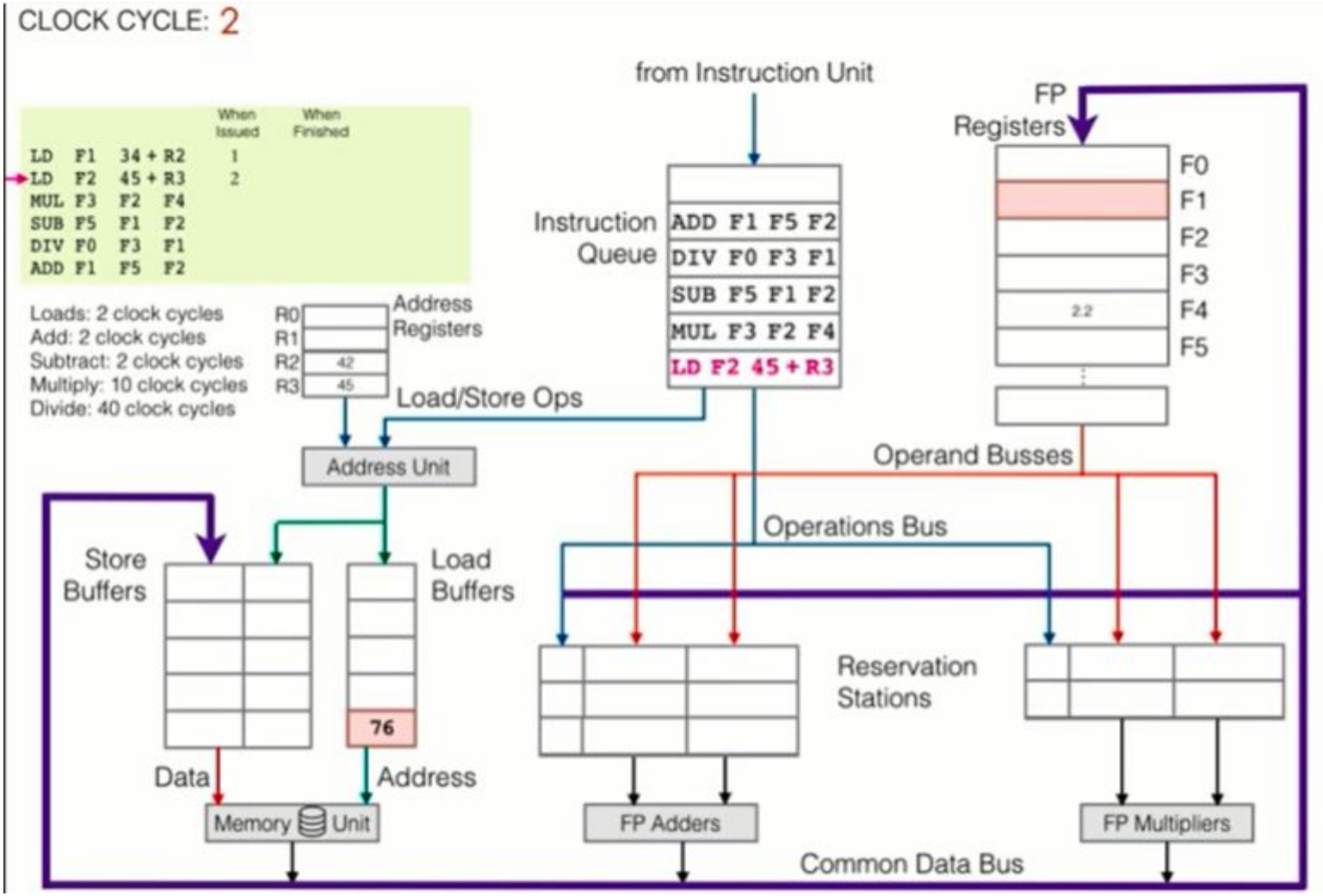
Algorithm



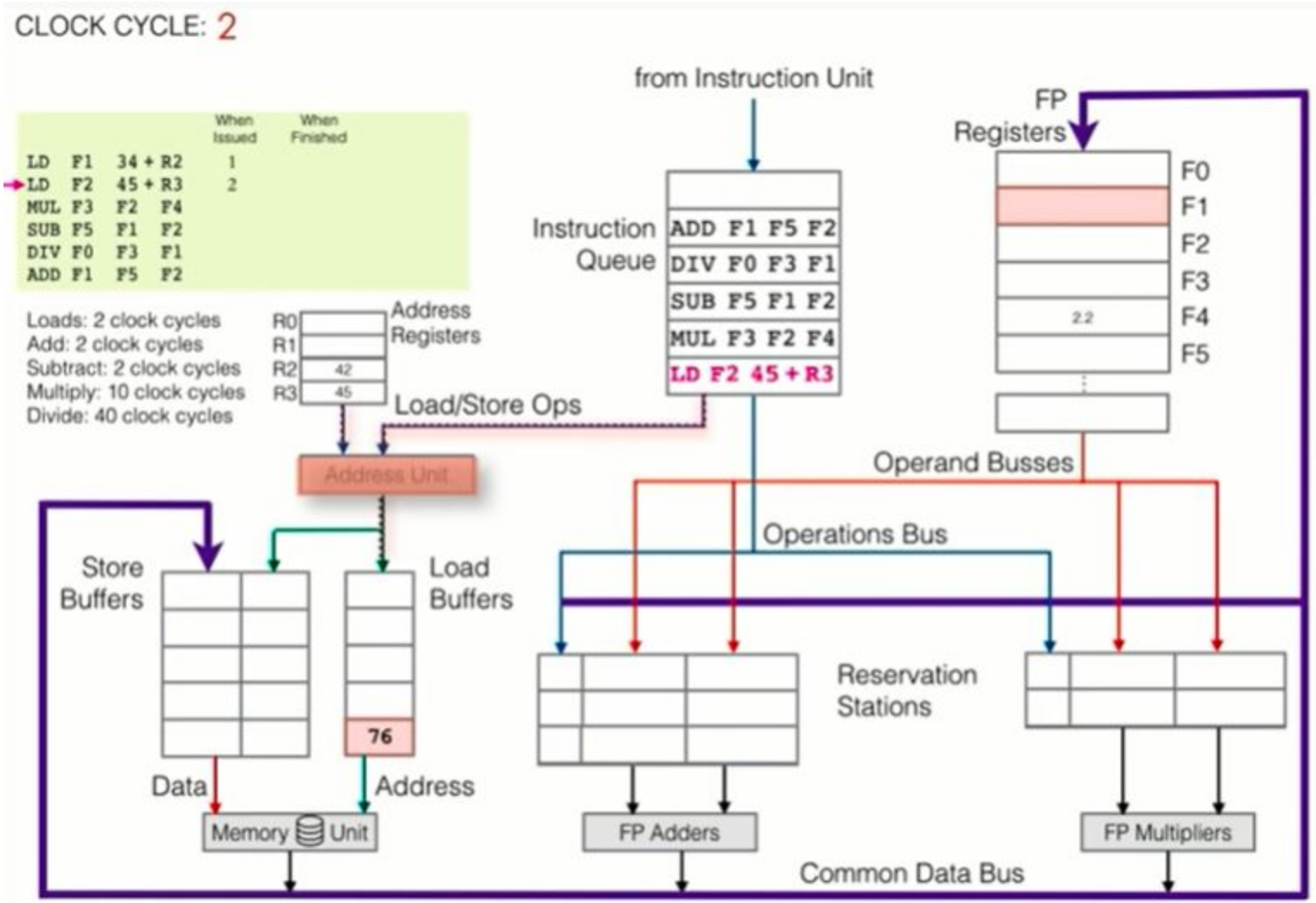
Algorithm



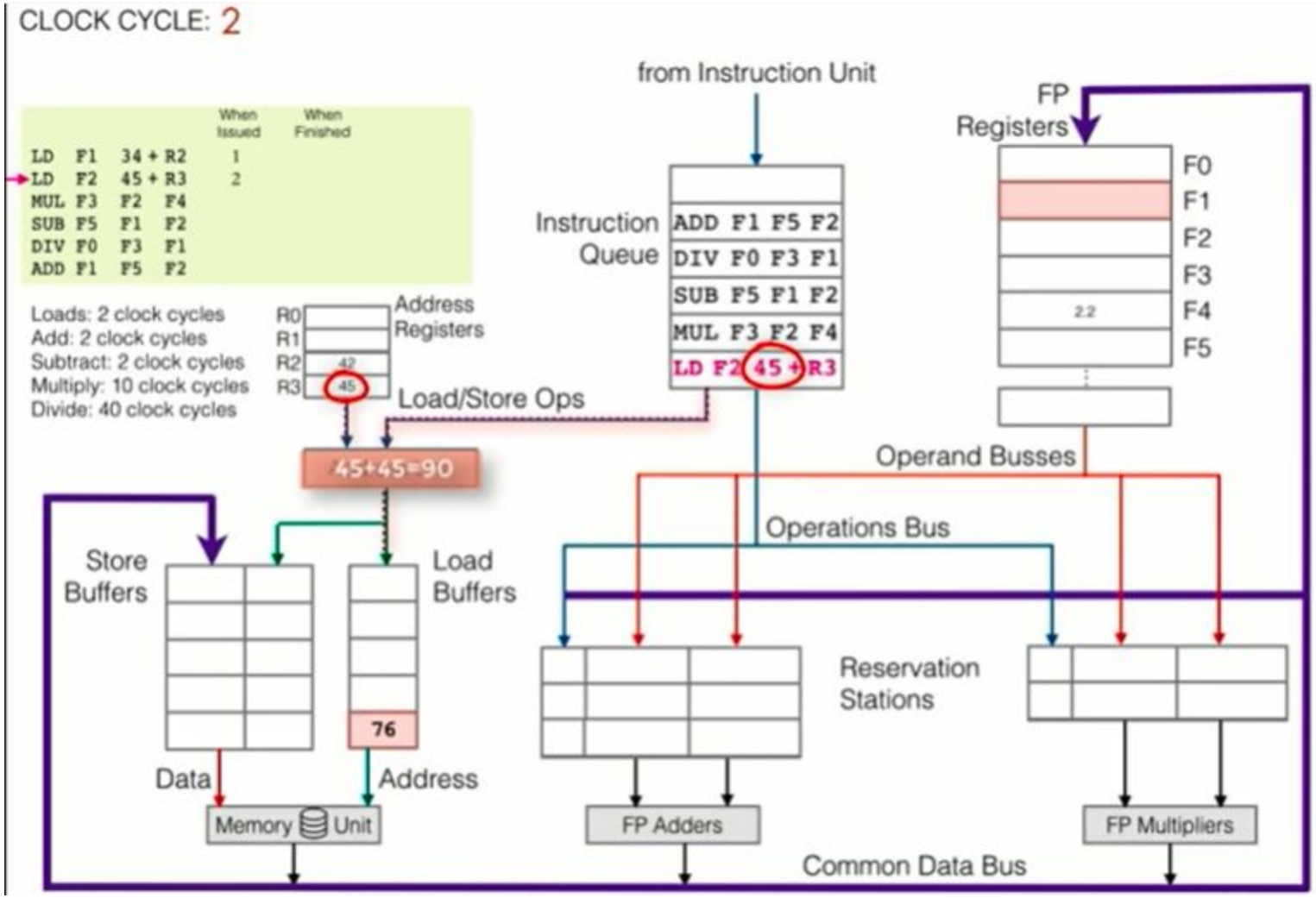
Algorithm



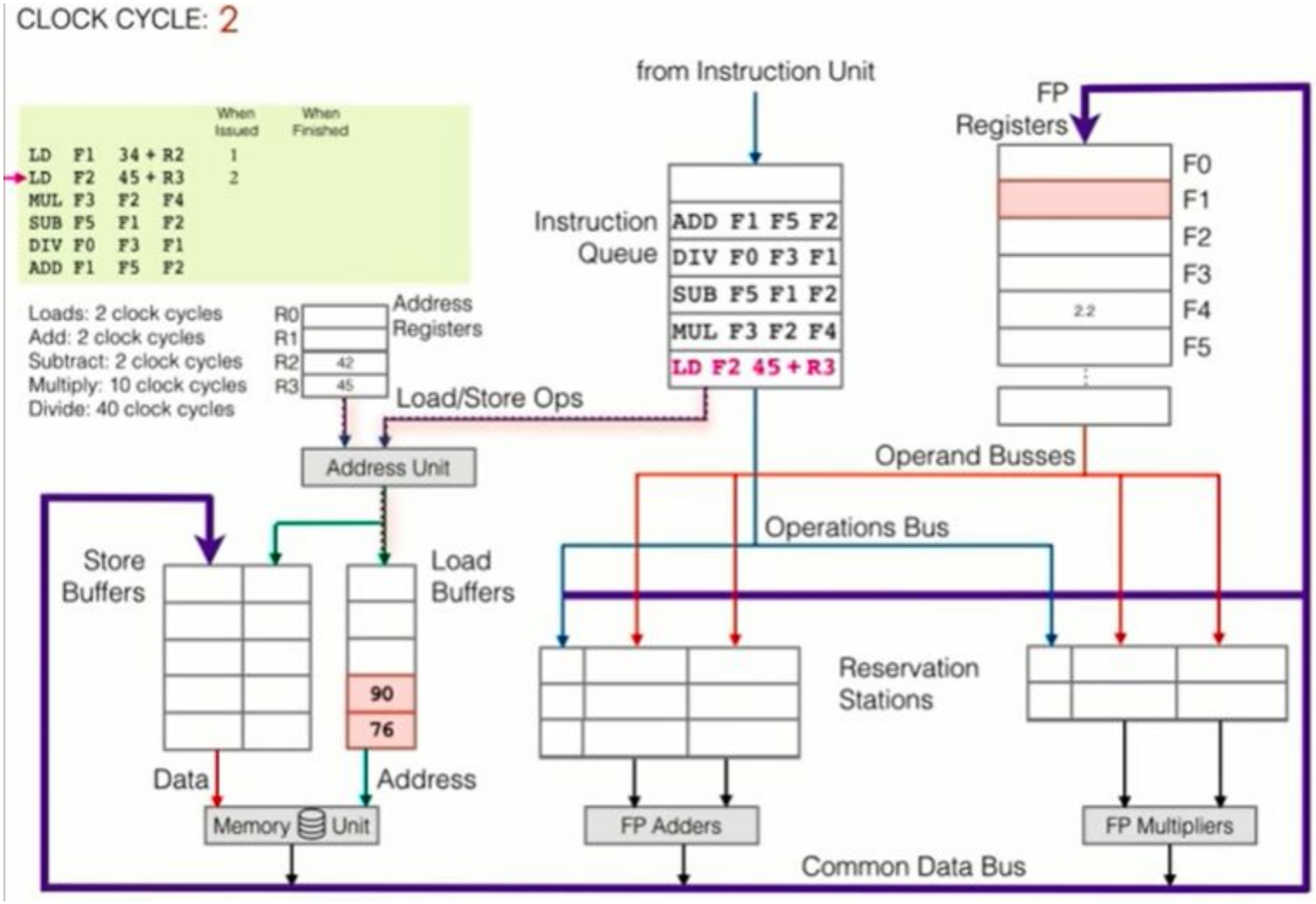
Algorithm



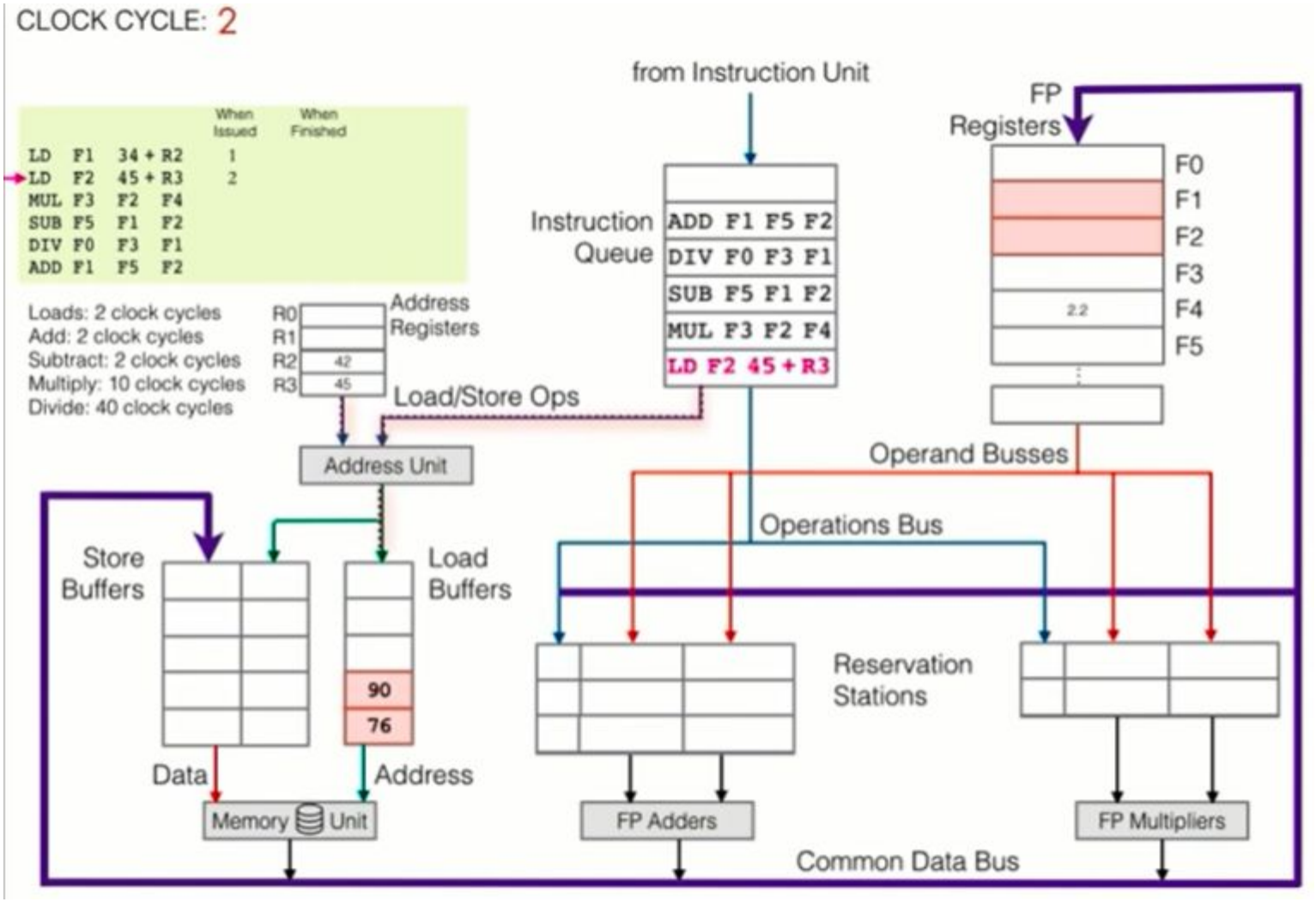
Dynamic Scheduling: Tomasulo's Algorithm



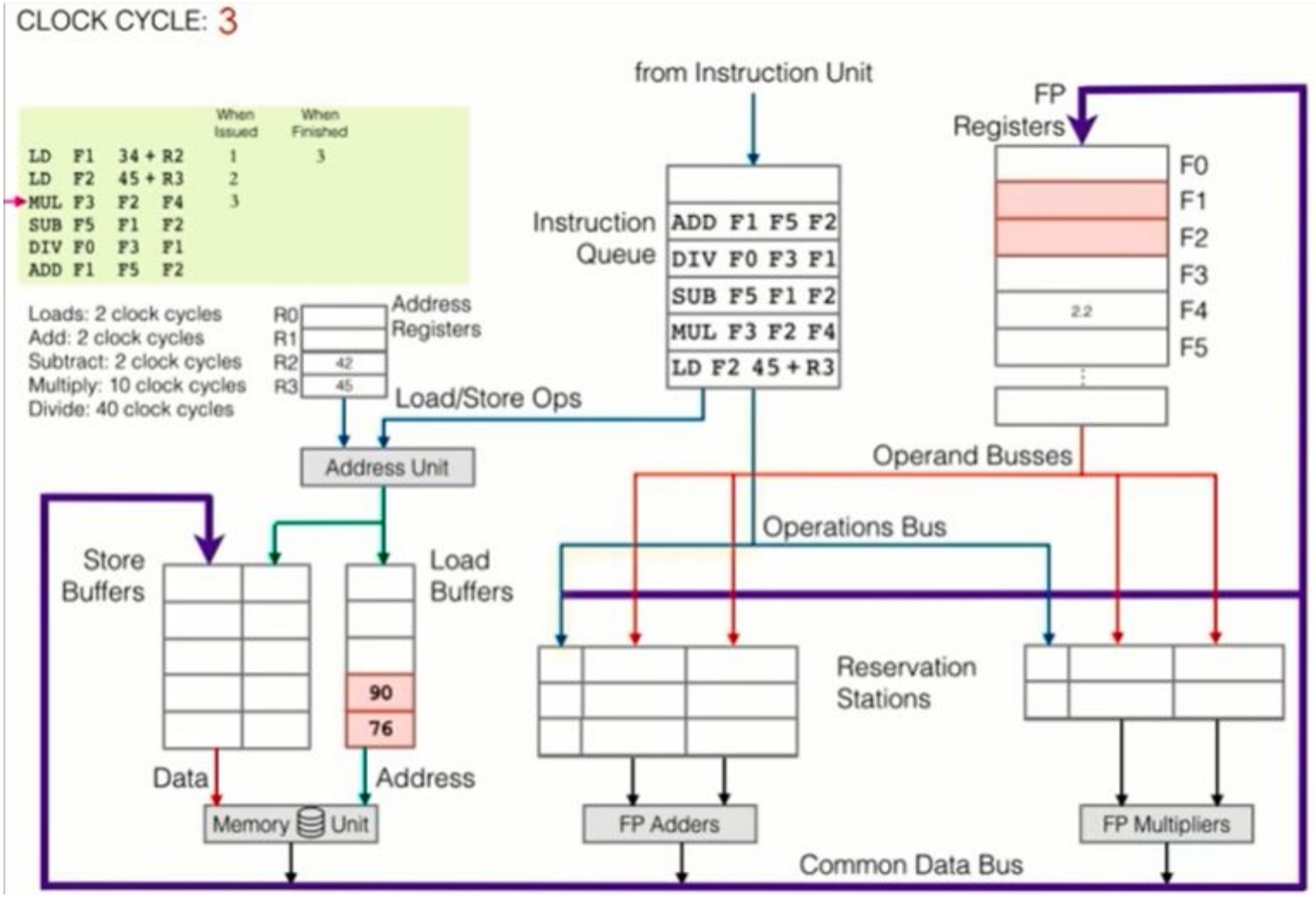
Dynamic Scheduling: Tomasulo's Algorithm



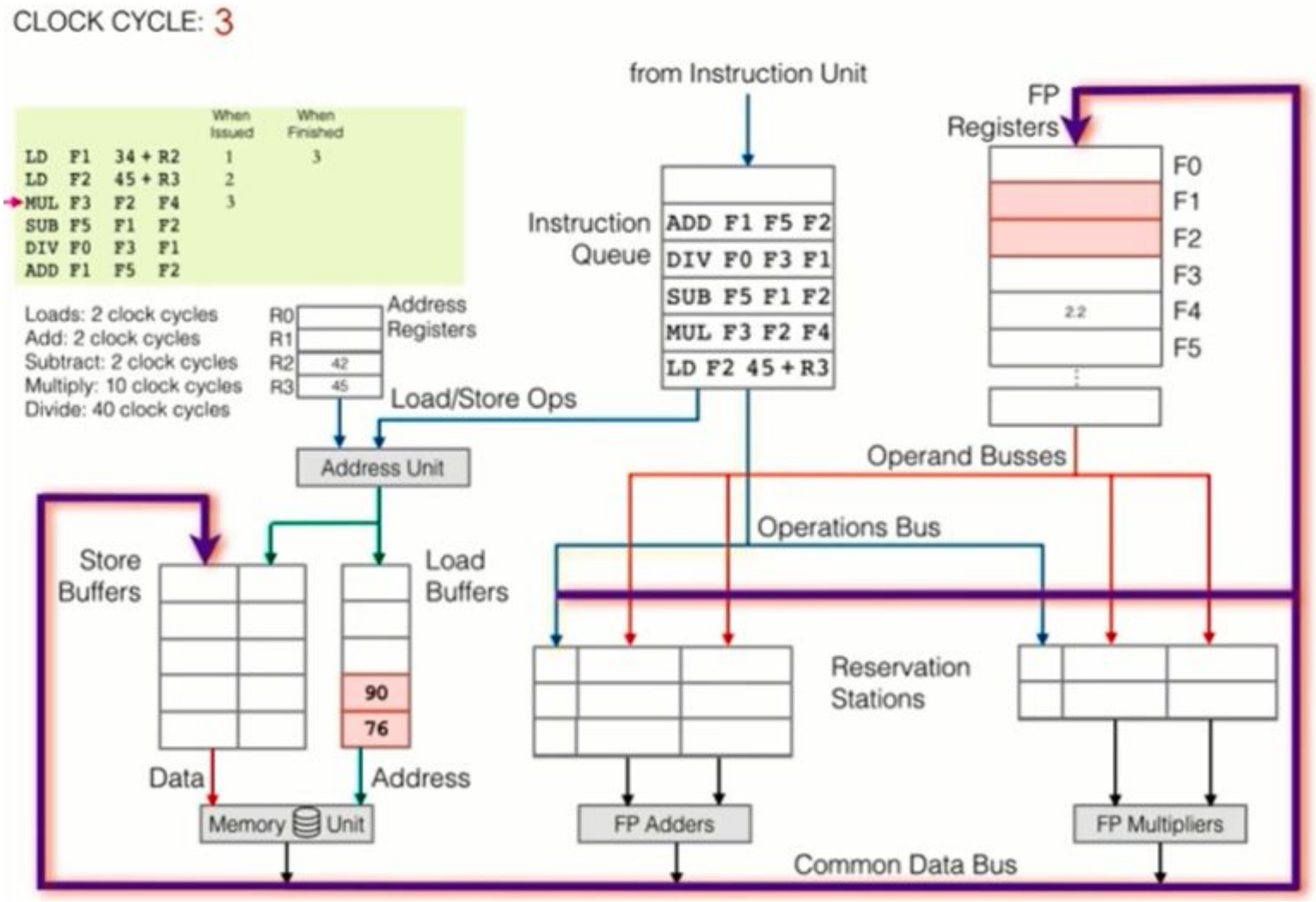
Algorithm



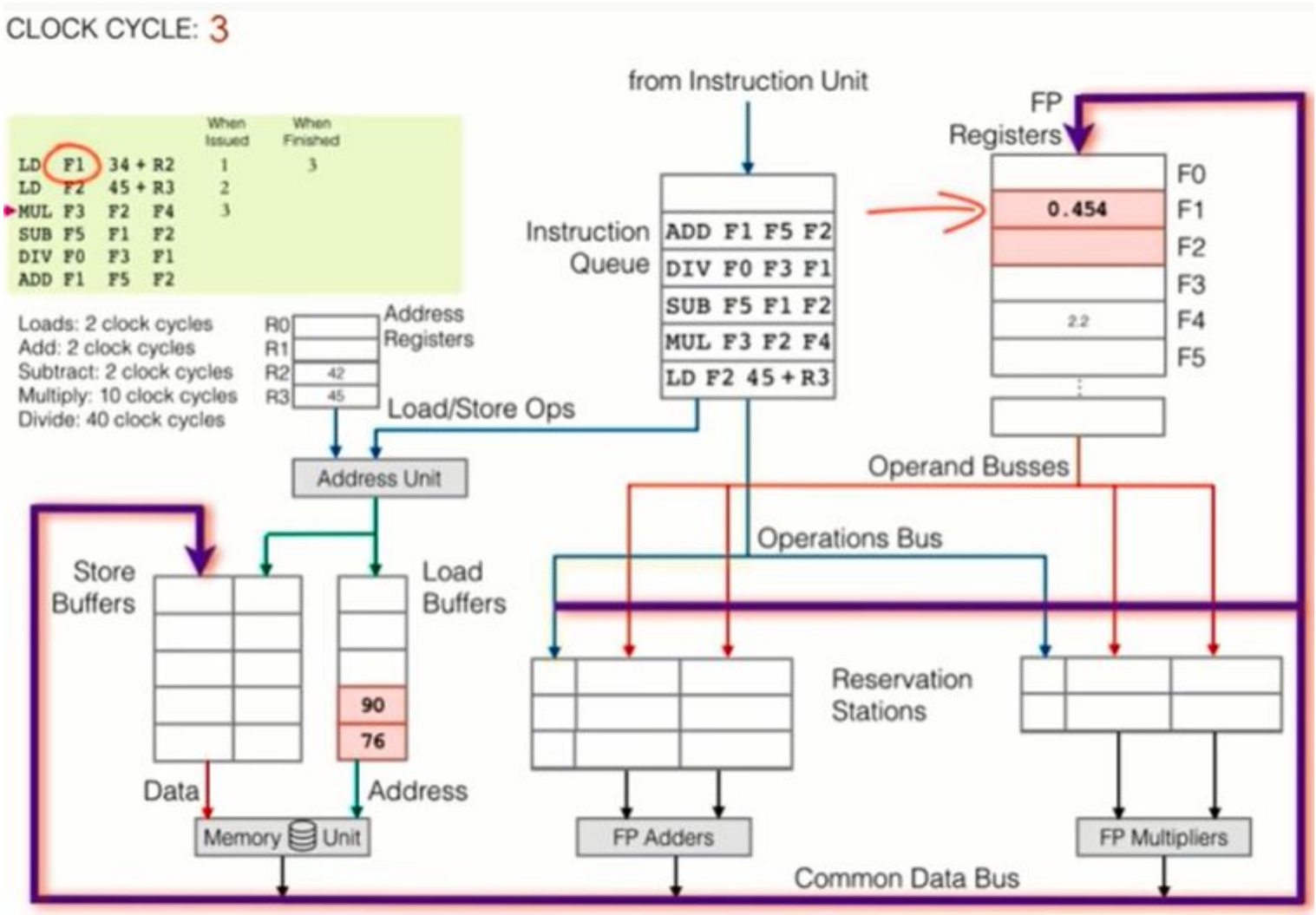
Algorithm



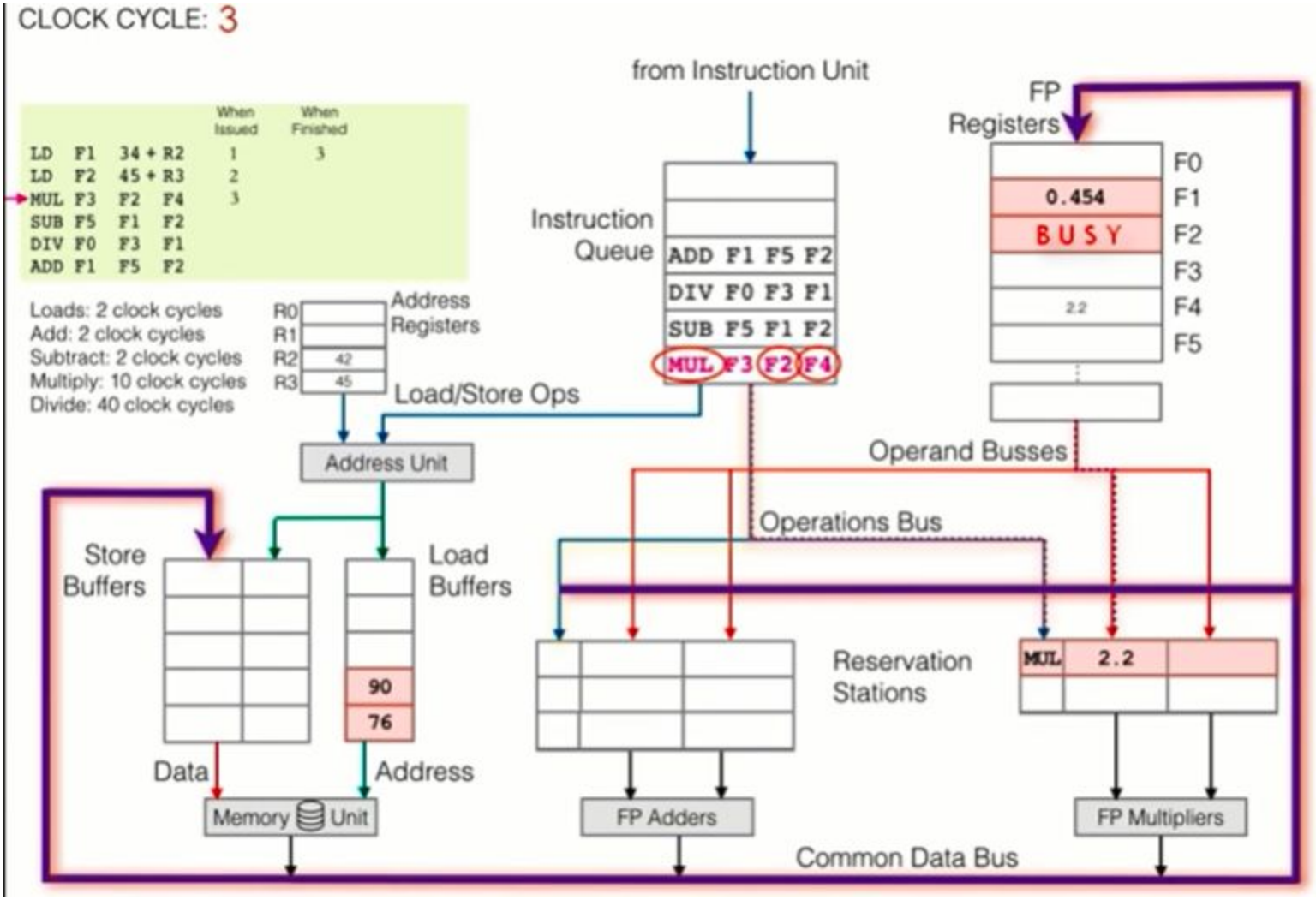
Algorithm



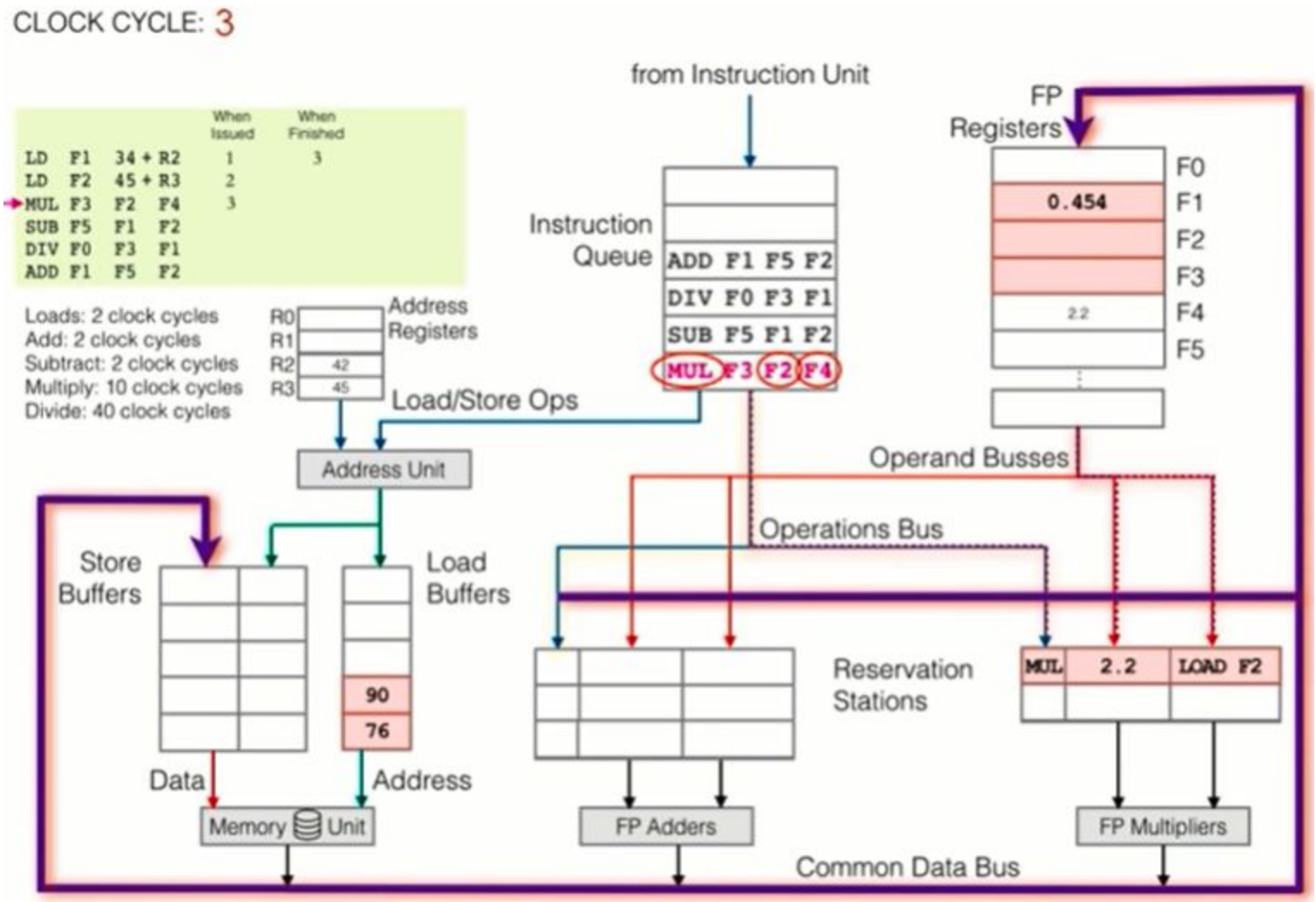
Algorithm



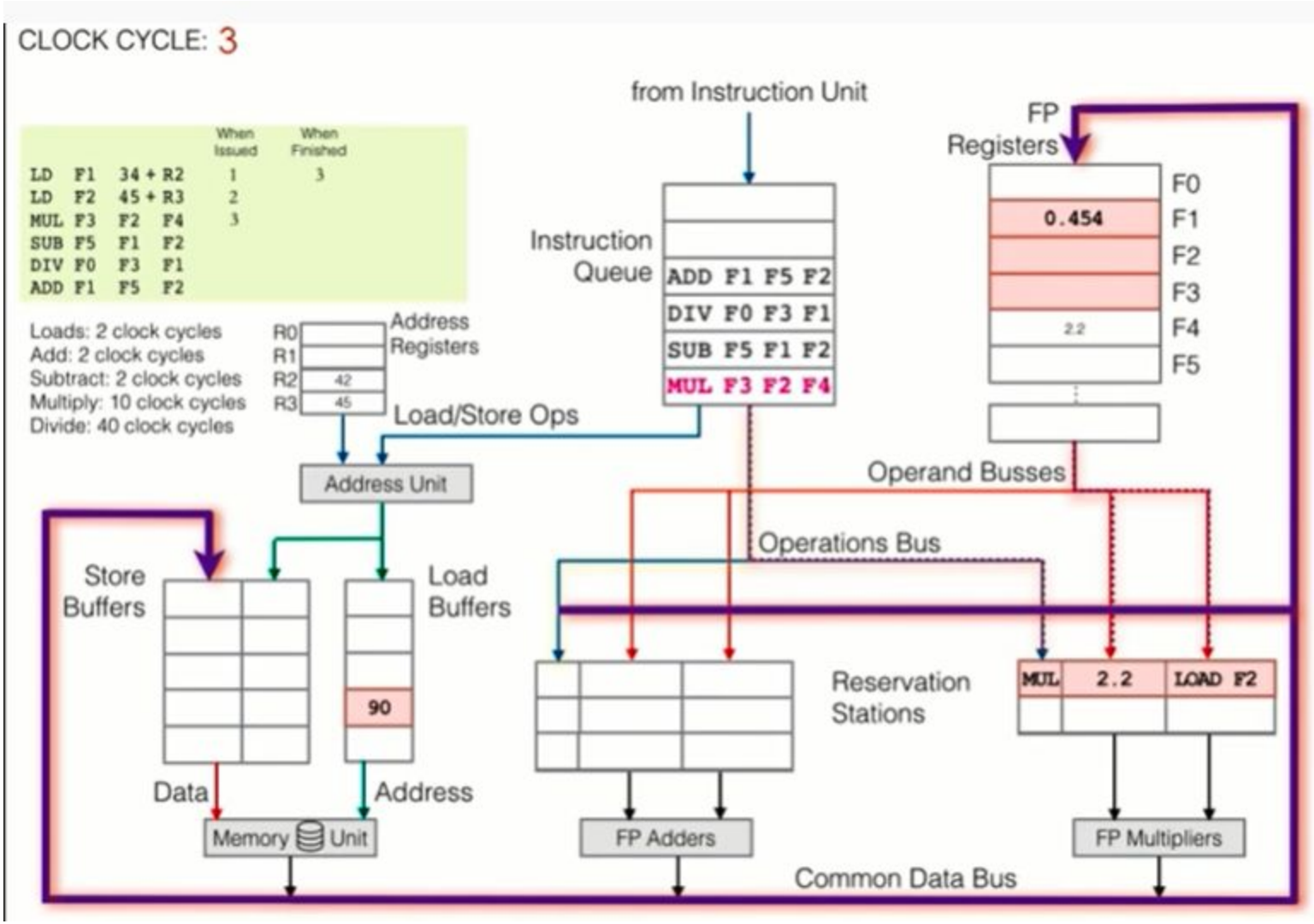
Algorithm



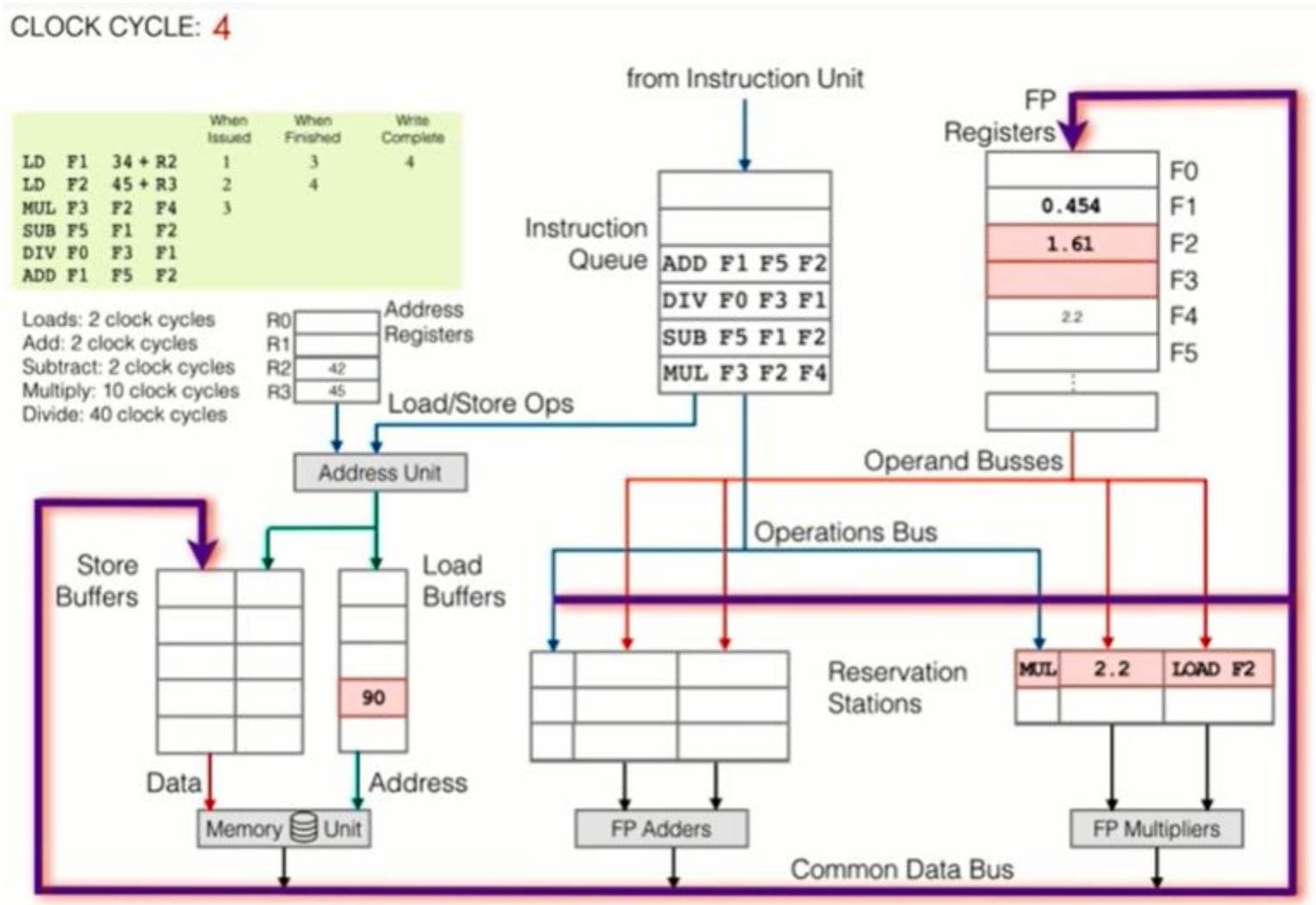
Algorithm



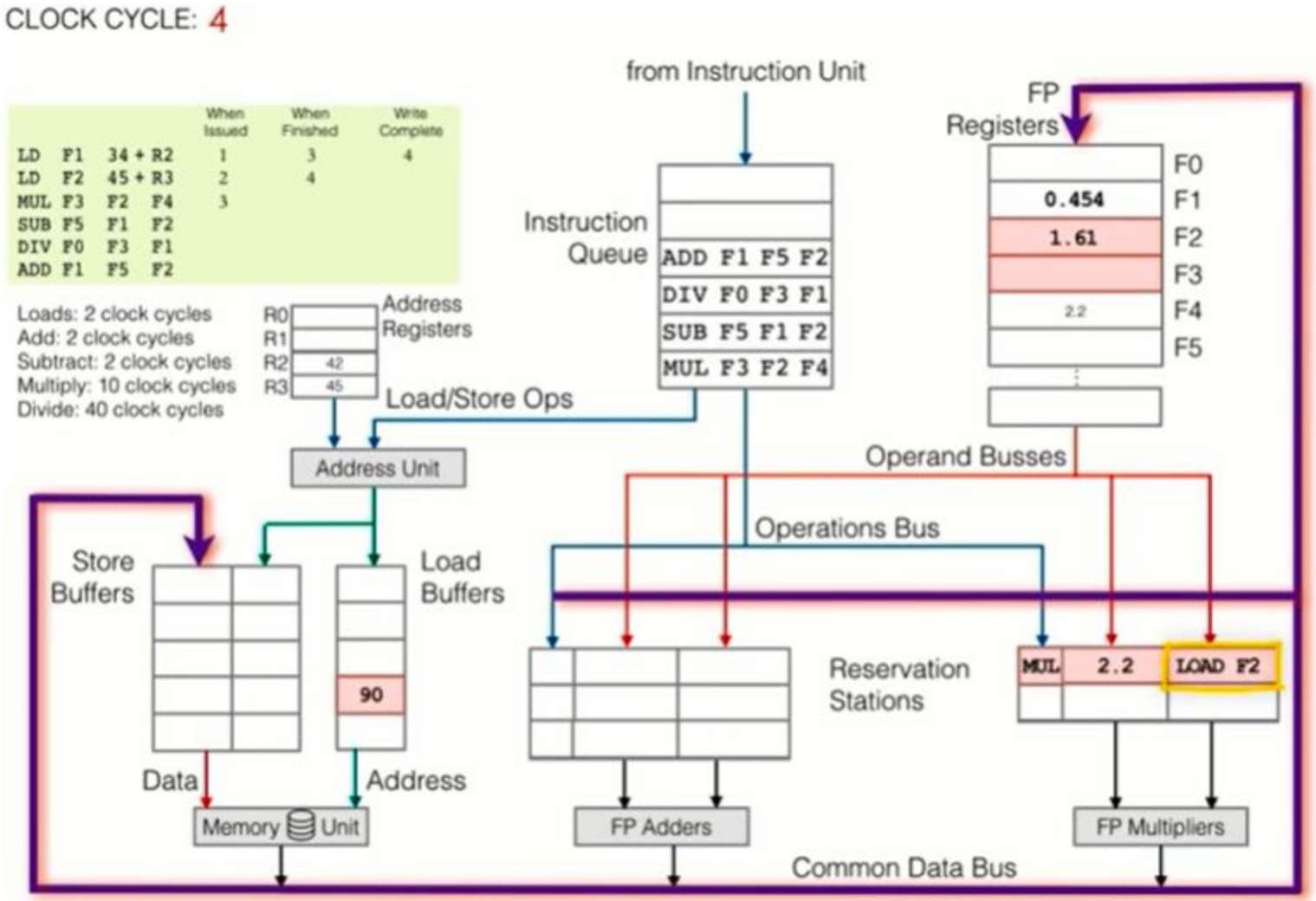
Algorithm



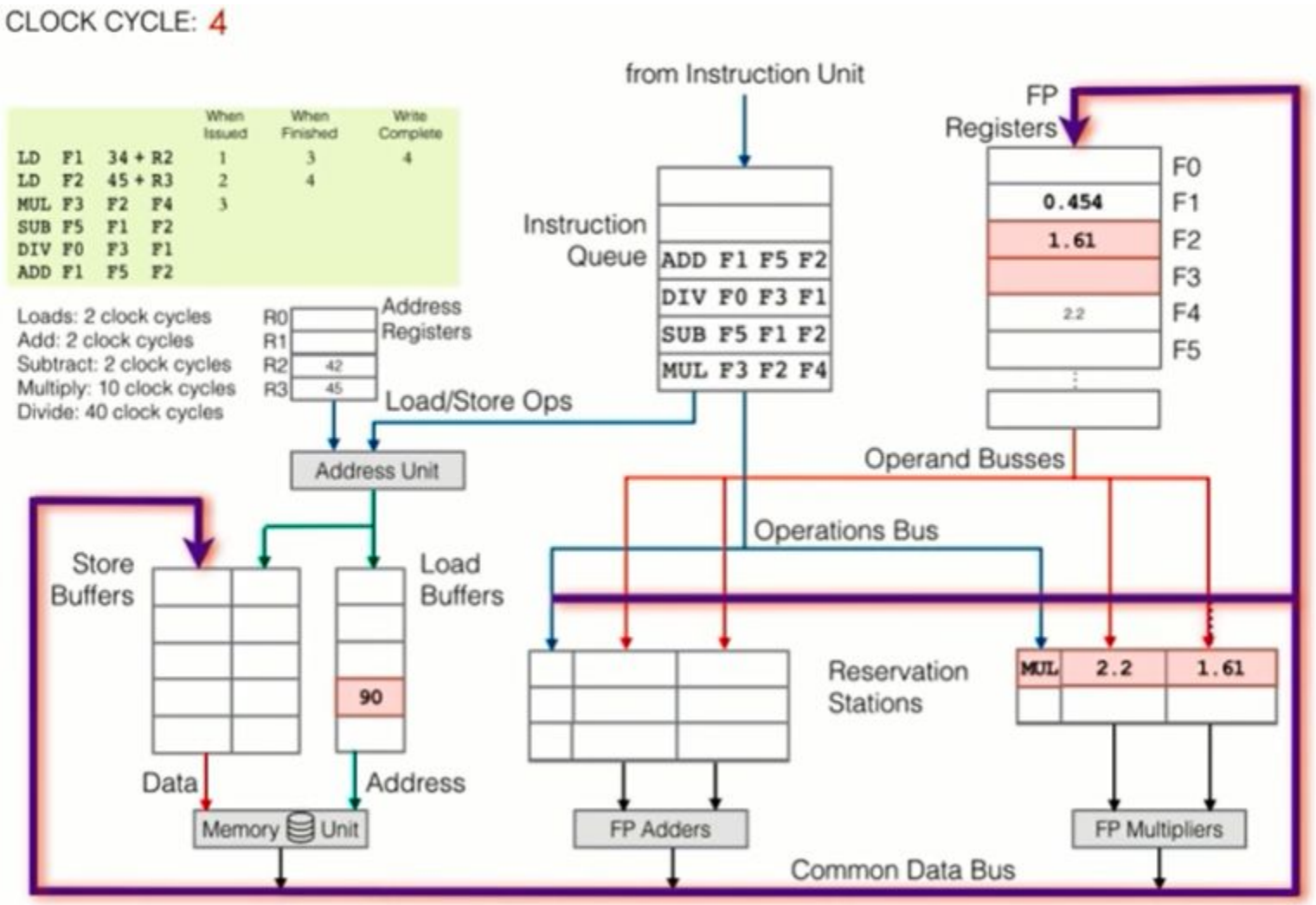
Dynamic Scheduling: Tomasulo's Algorithm



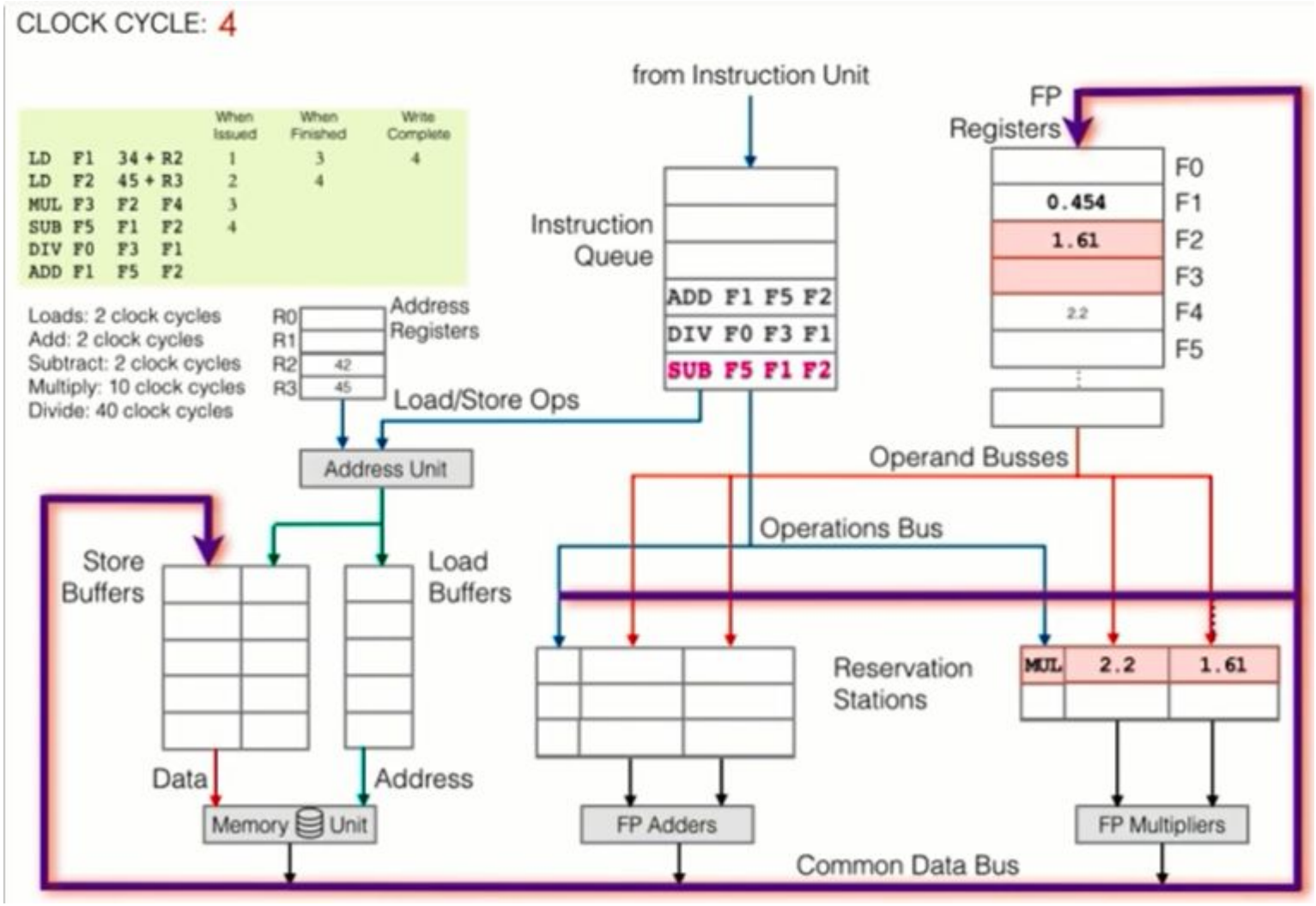
Algorithm



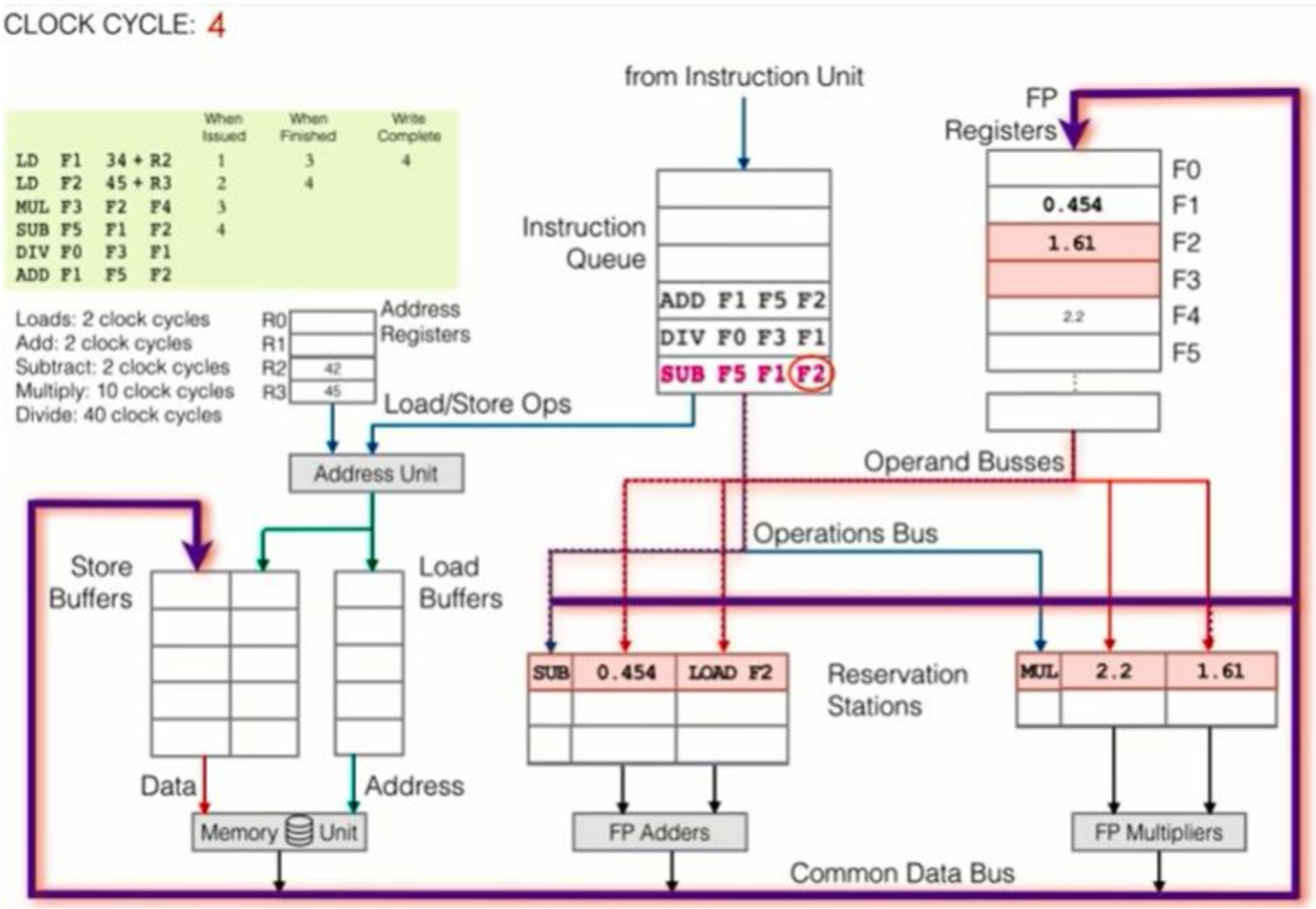
Dynamic Scheduling: Tomasulo's Algorithm



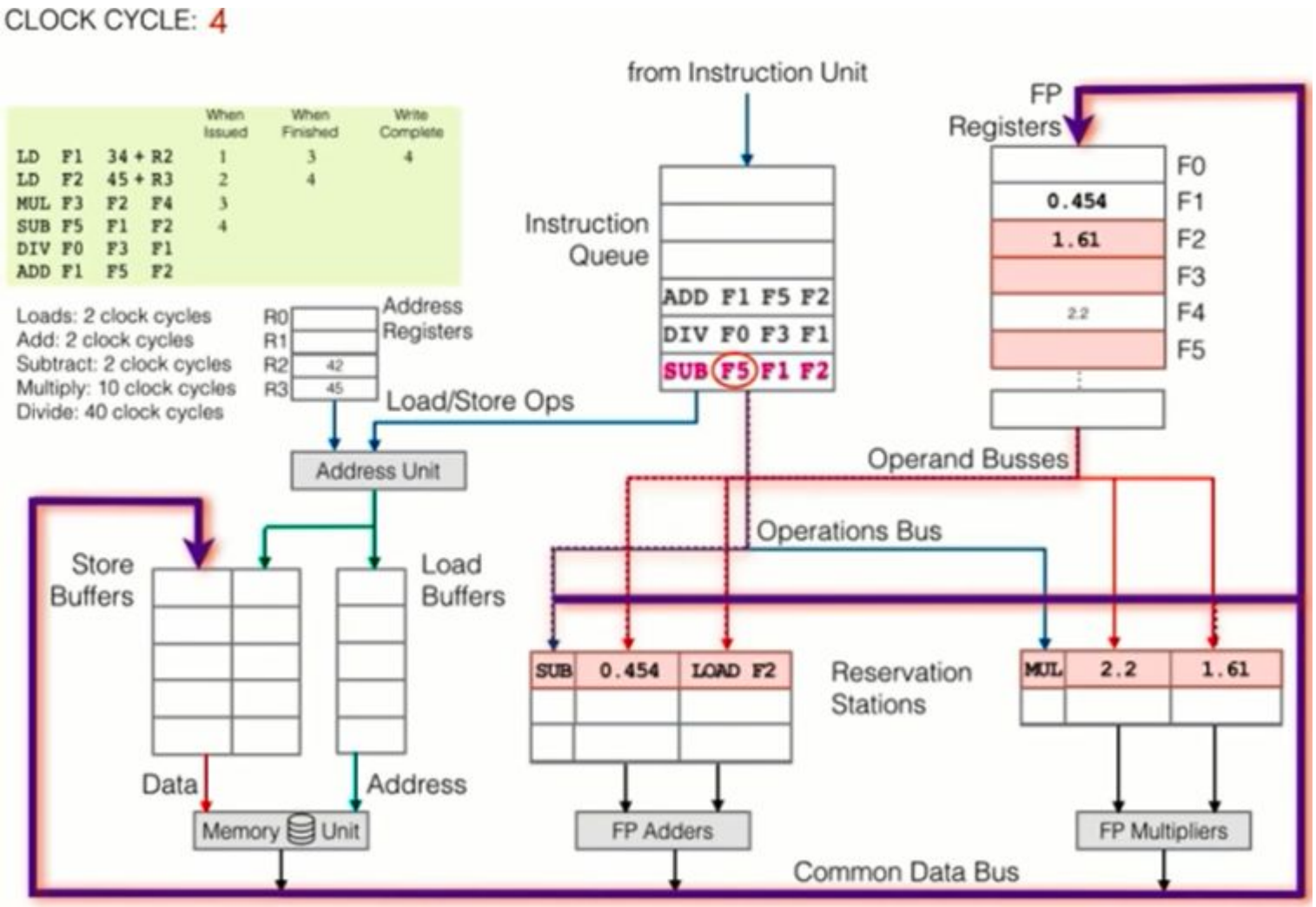
Algorithm



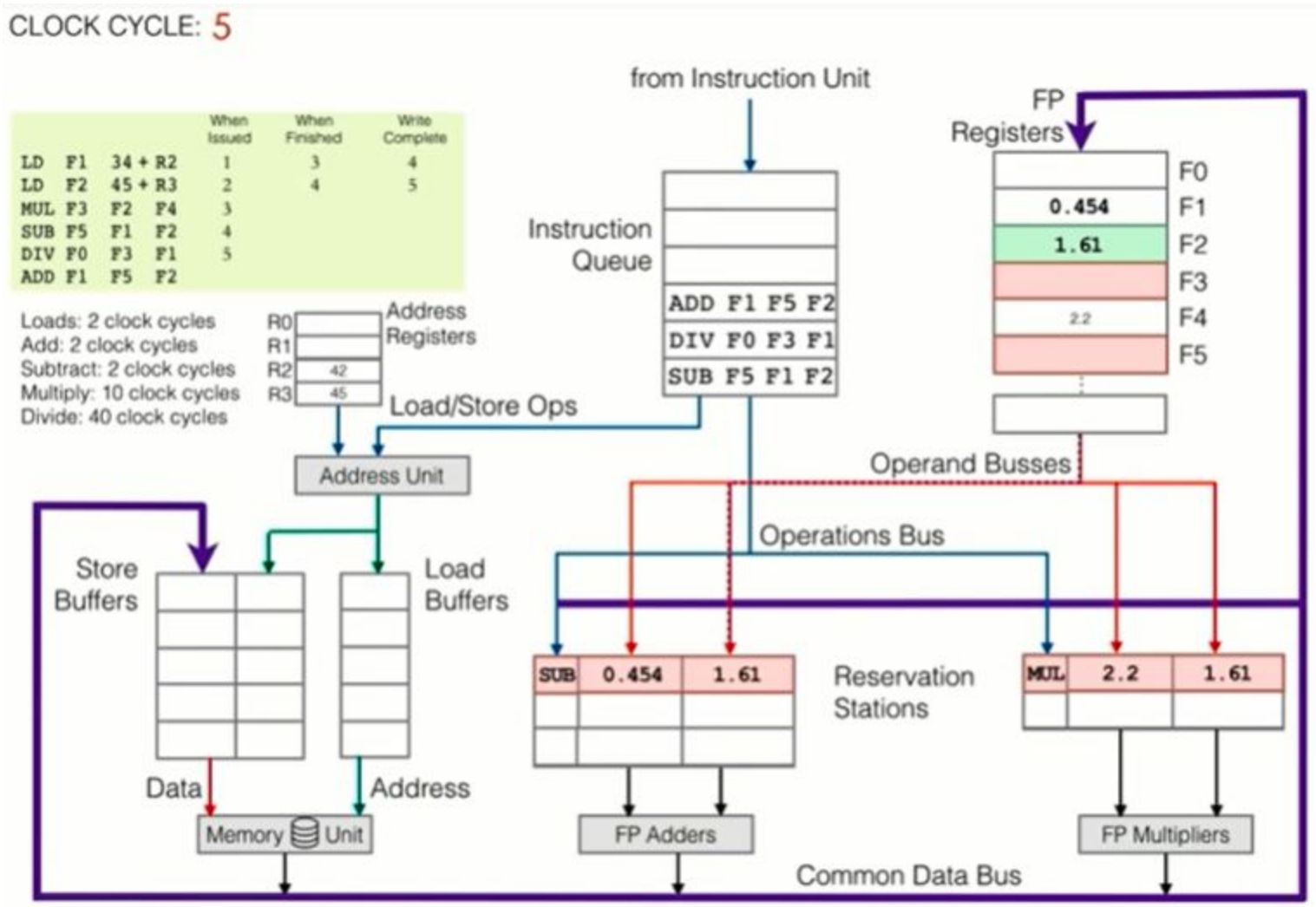
Algorithm



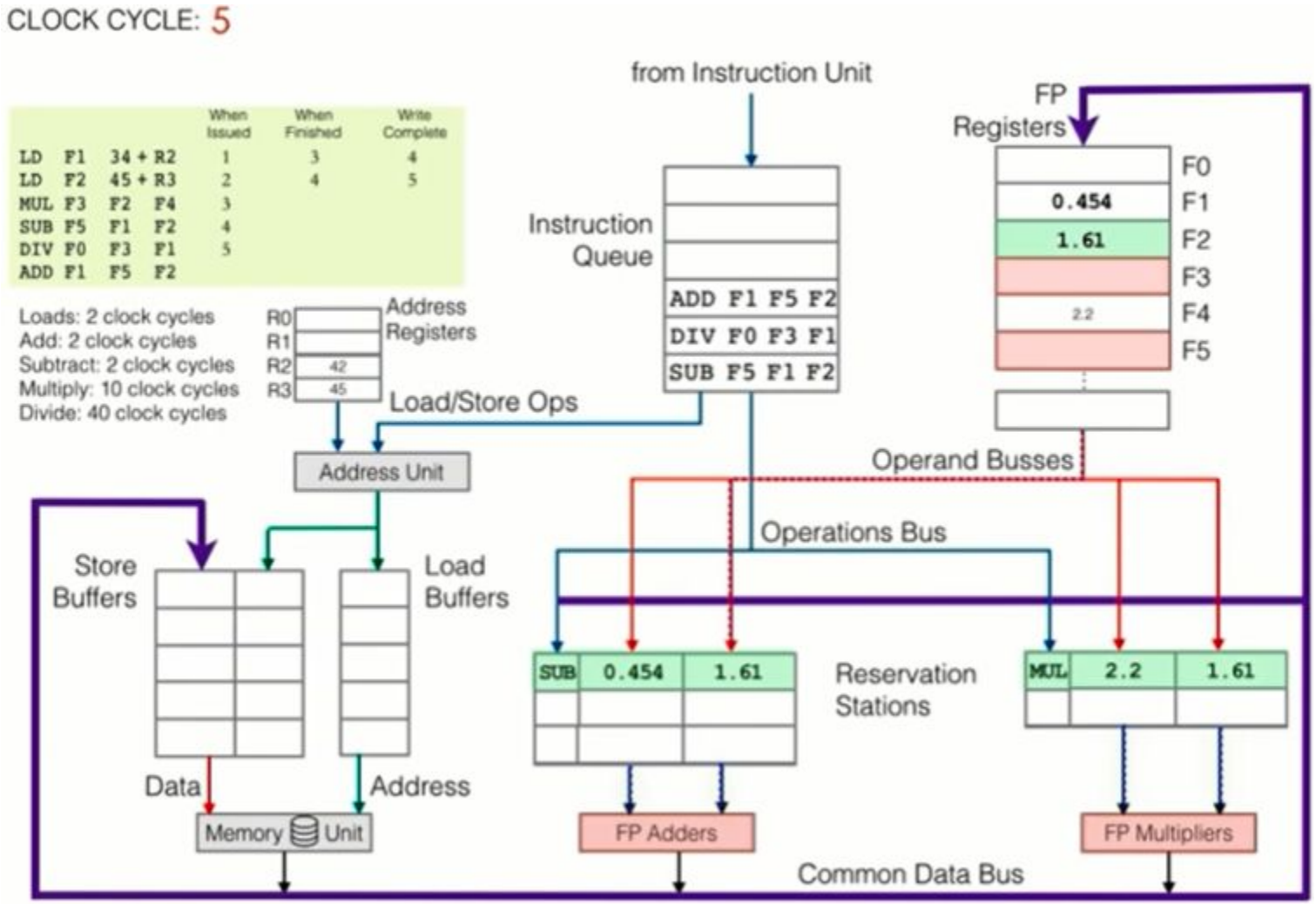
Algorithm



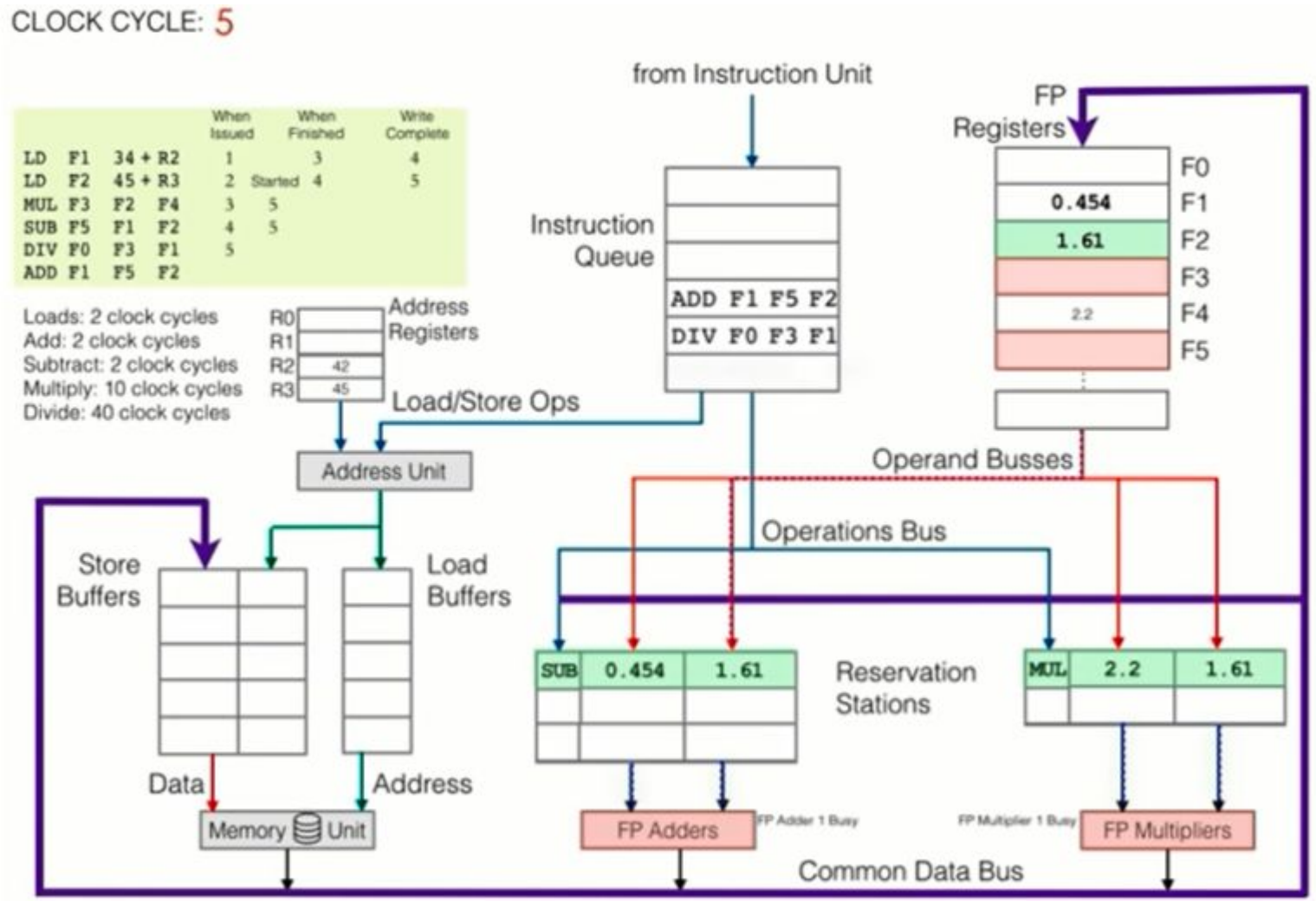
Algorithm



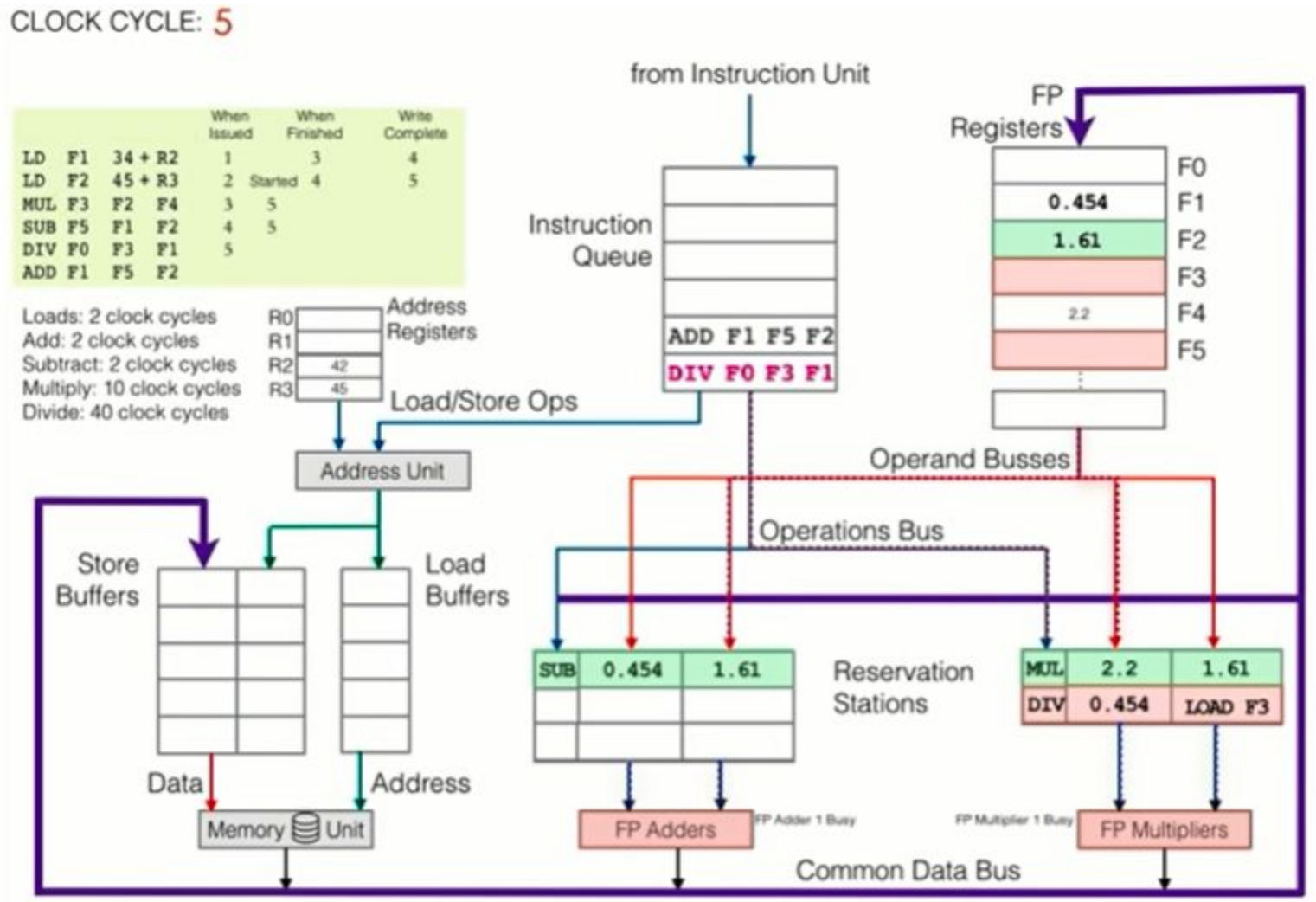
Algorithm



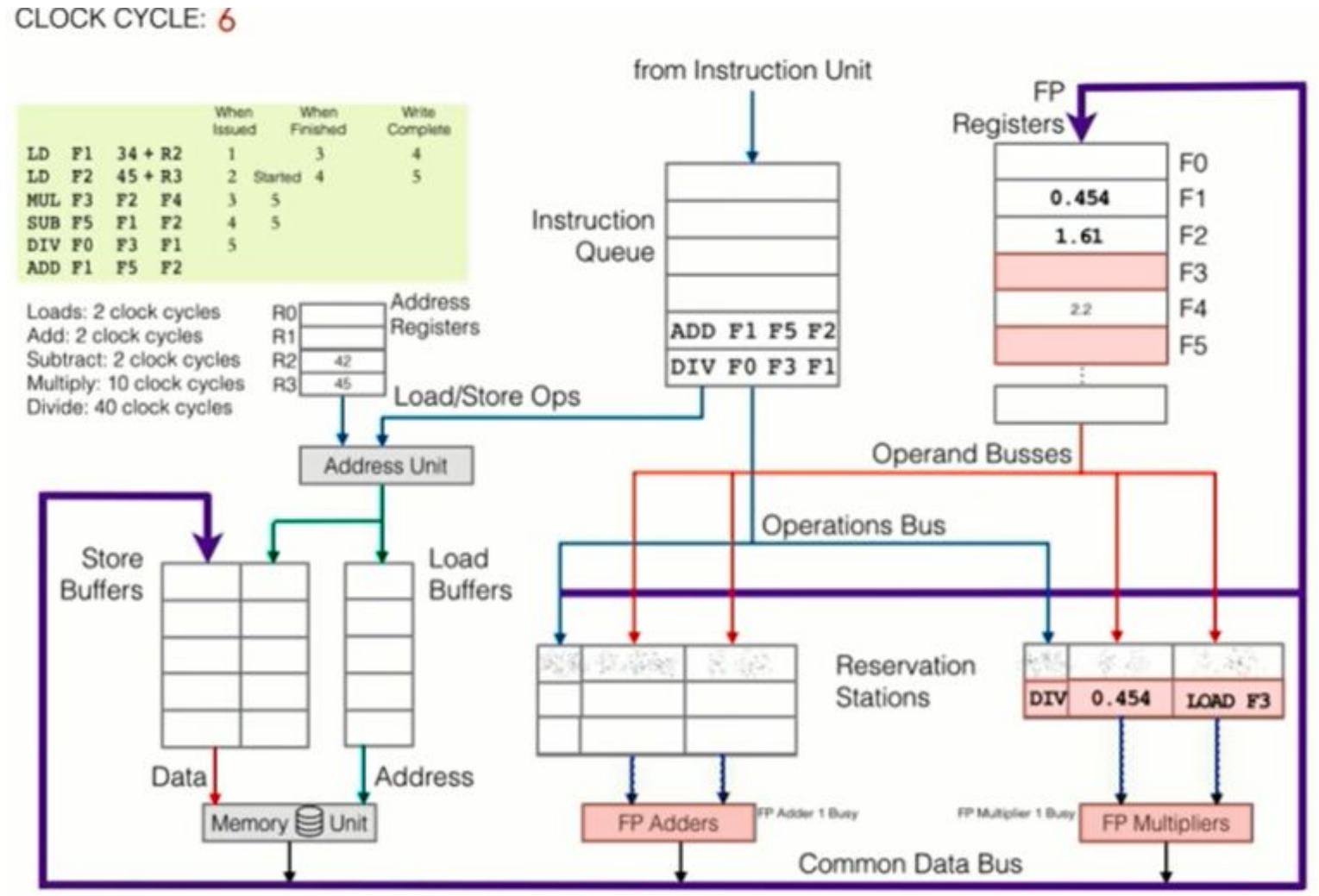
Algorithm



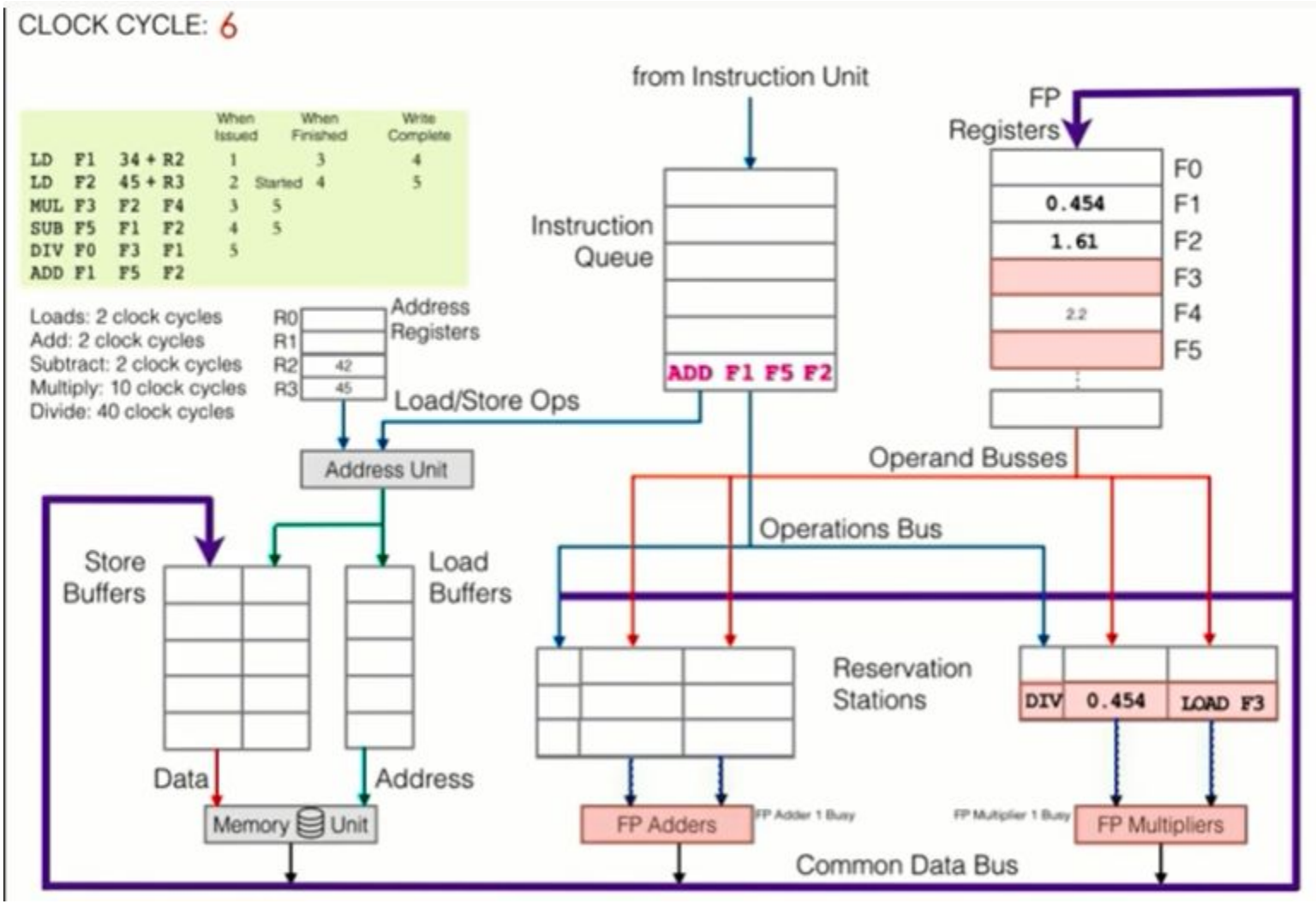
Algorithm



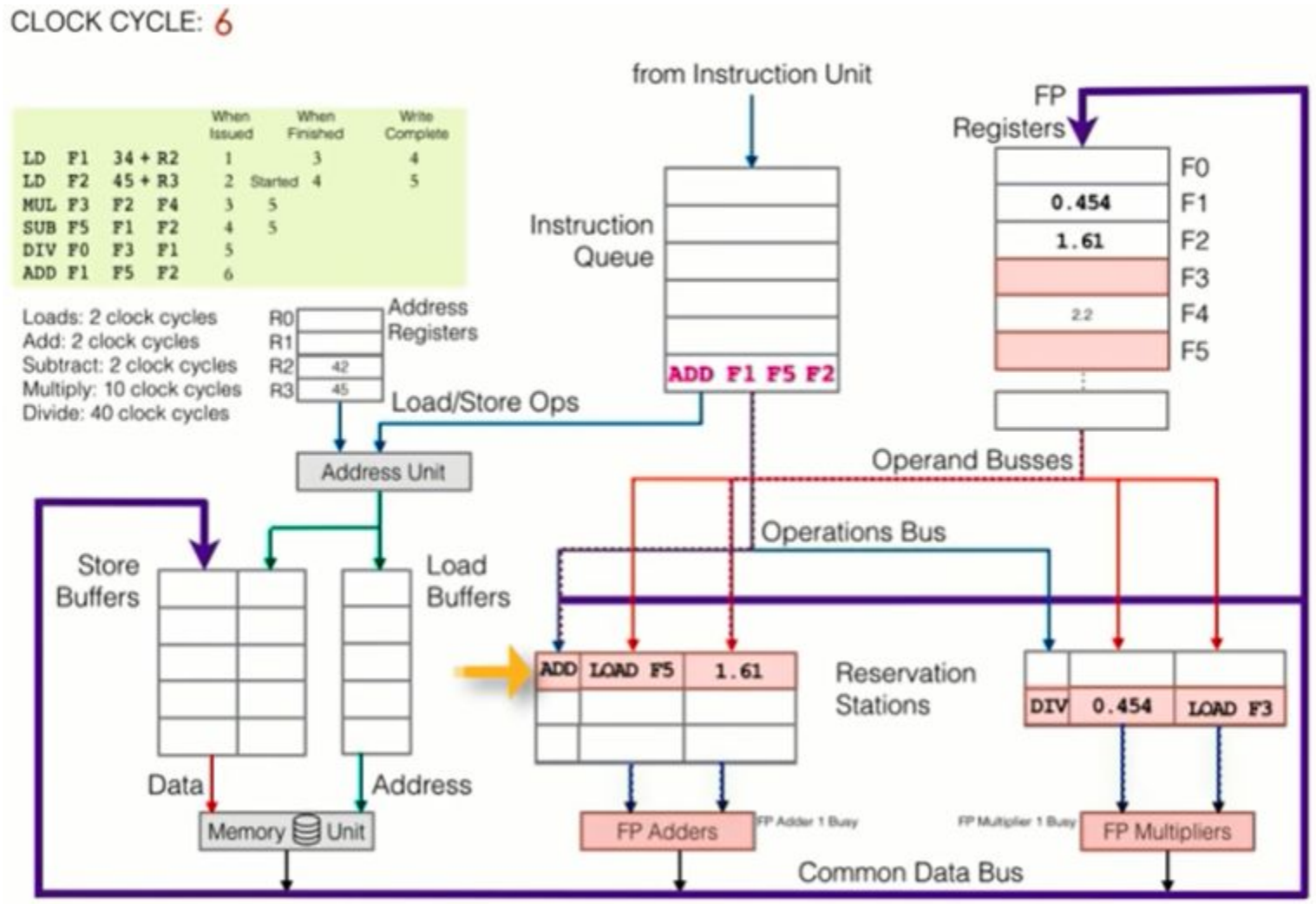
Algorithm



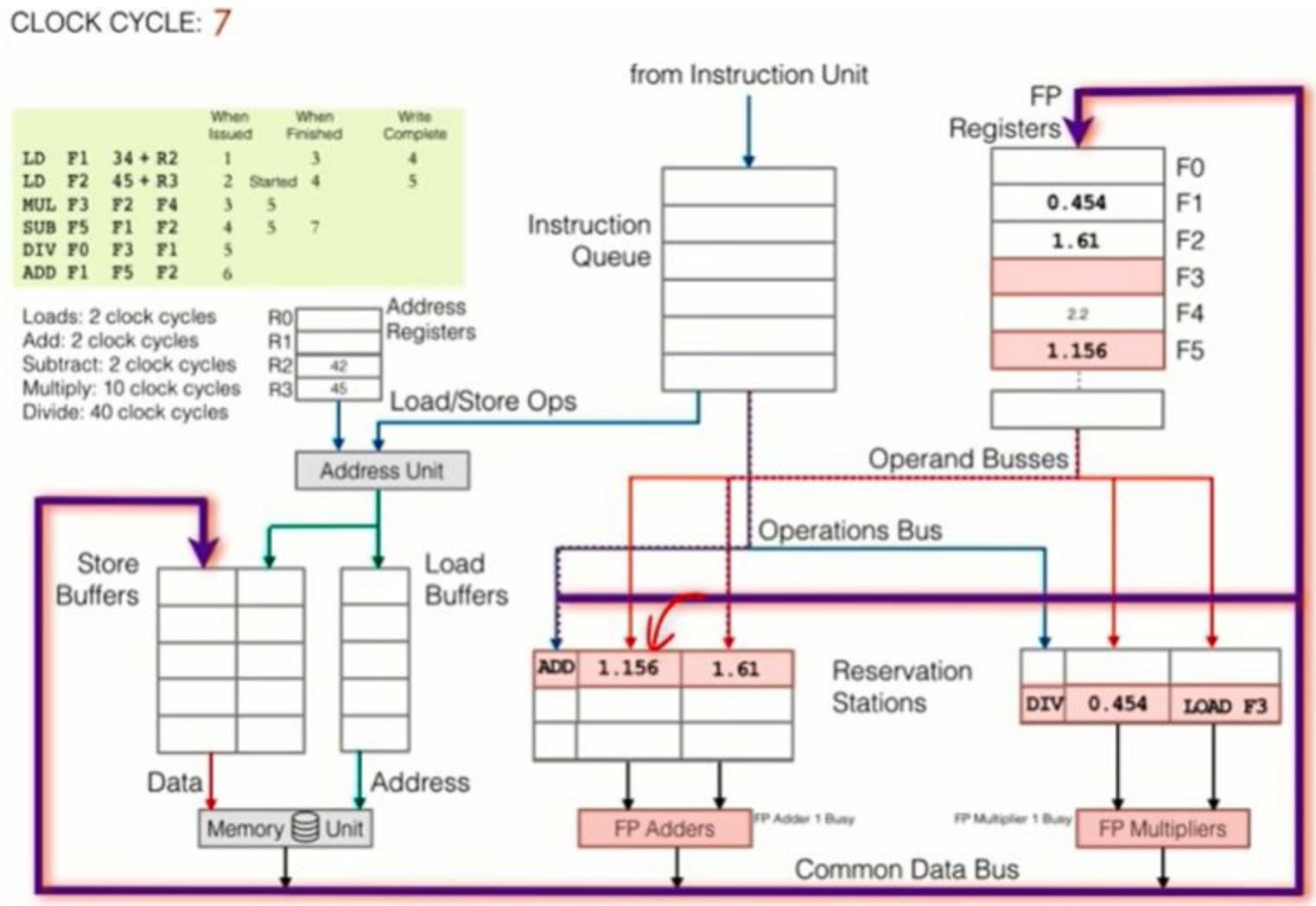
Algorithm



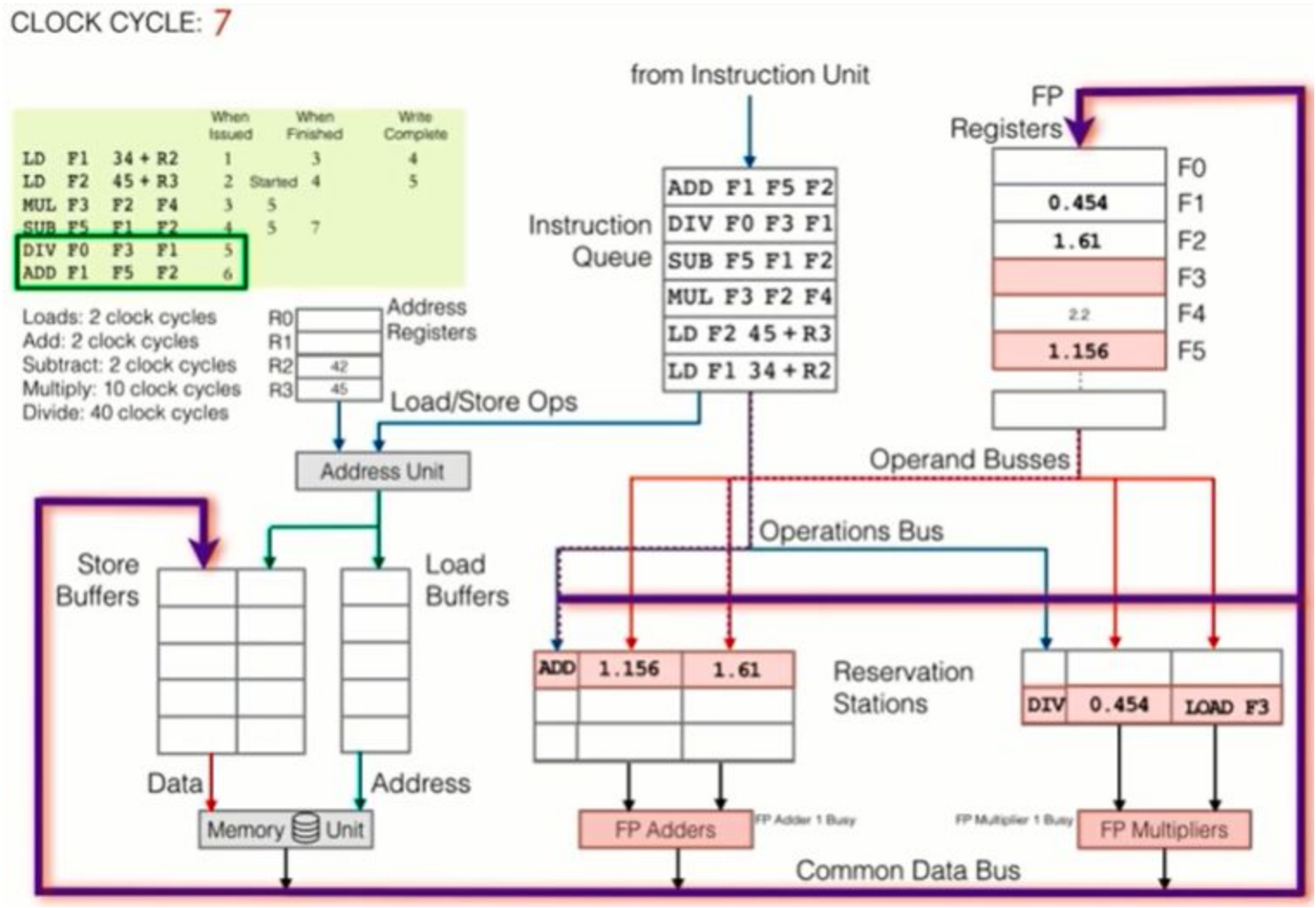
Algorithm



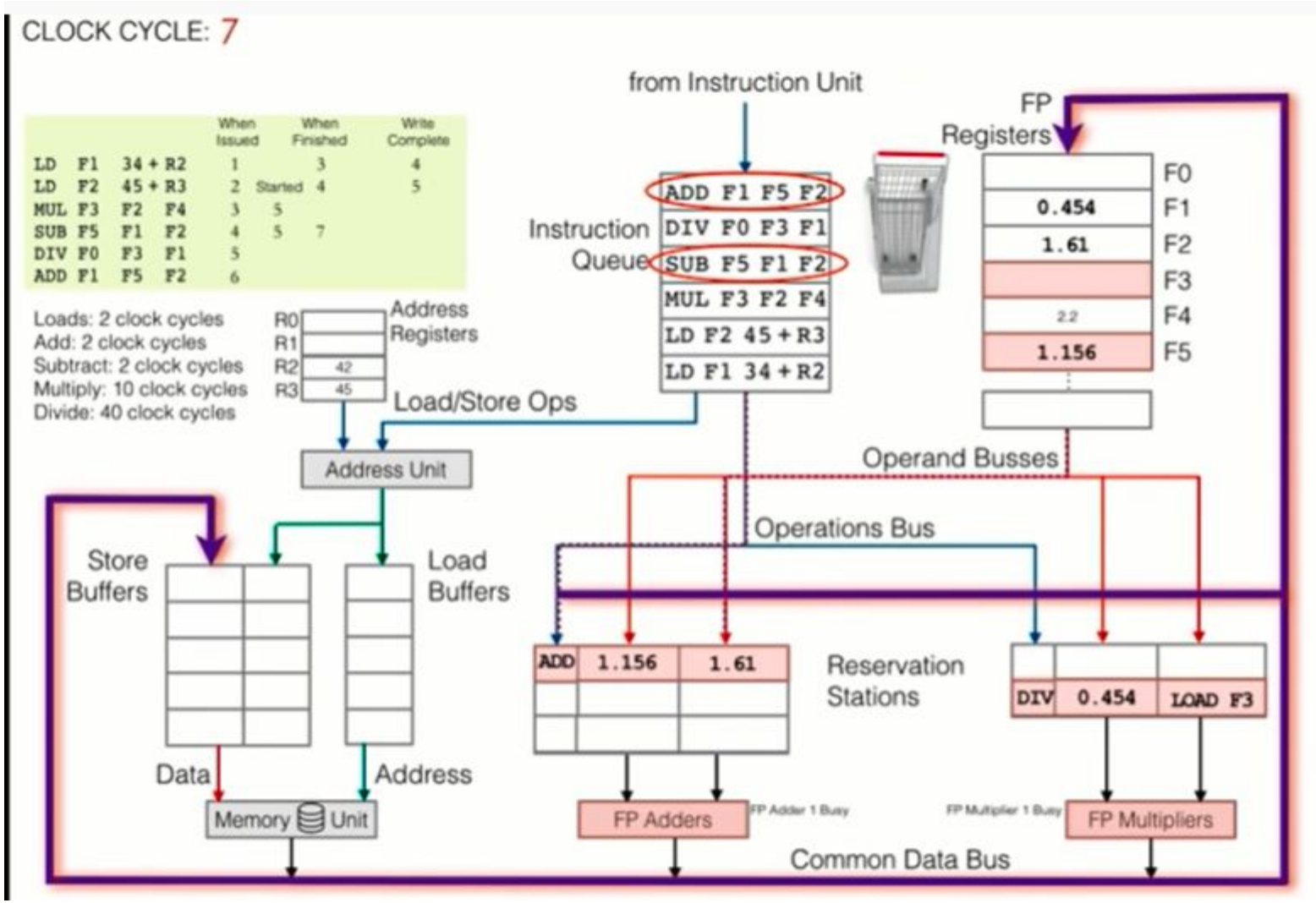
Algorithm



Algorithm



Algorithm

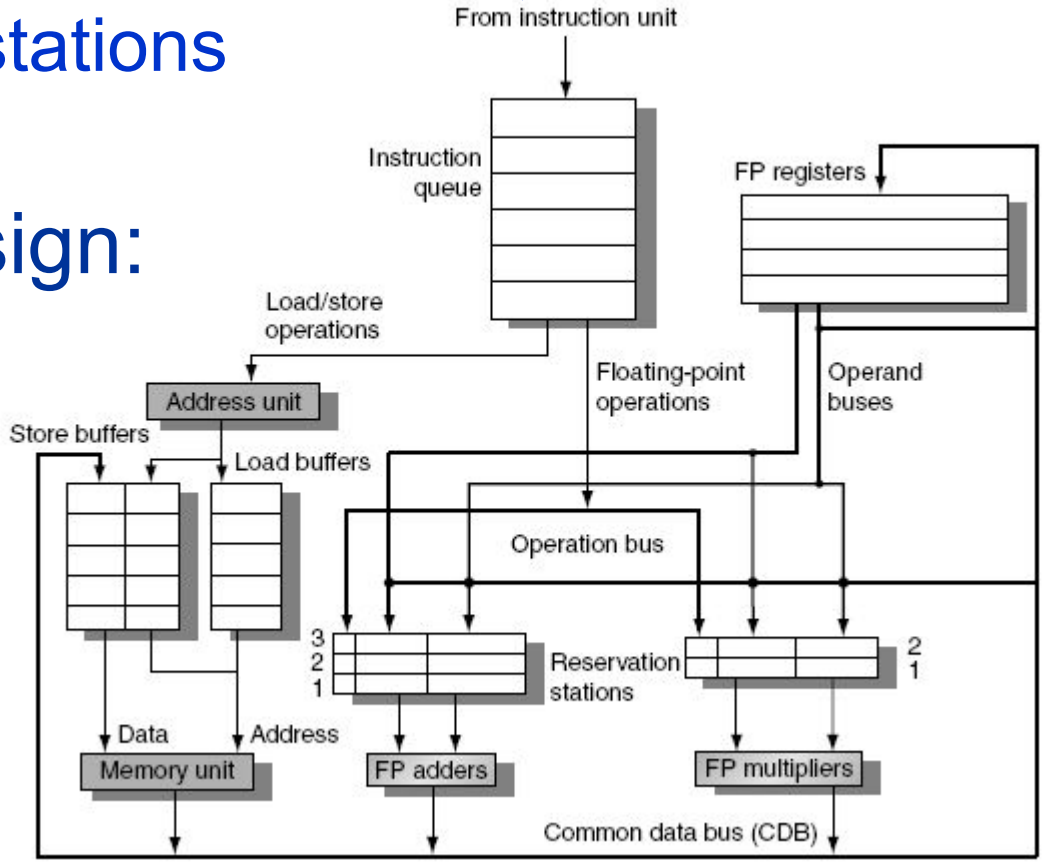


Register Renaming

- Register renaming is provided by reservation stations (RS)
  - Contains:
    - The instruction
    - Buffered operand values (when available)
    - Reservation station number of instruction providing the operand values
  - RS fetches and buffers an operand as soon as it becomes available (not necessarily involving register file)
  - Pending instructions designate the RS to which they will send their output
    - Result values broadcast on a result bus, called the common data bus (CDB)
  - Only the last output updates the register file
  - As instructions are issued, the register specifiers are renamed with the reservation station
  - May be more reservation stations than registers

Tomasulo's Algorithm

- Load and store buffers
  - Contain data and addresses, act like reservation stations
- Top-level design:



Tomasulo's Algorithm

- Three Steps:
  - Issue
    - Get next instruction from FIFO queue
    - If available RS, issue the instruction to the RS with operand values if available
    - If operand values not available, stall the instruction
  - Execute
    - When operand becomes available, store it in any reservation stations waiting for it
    - When all operands are ready, issue the instruction
    - Loads and store maintained in program order through effective address
    - No instruction allowed to initiate execution until all branches that proceed it in program order have completed
  - Write result
    - Write result on CDB into reservation stations and store buffers
      - (Stores must wait until address and value are received)

Example

		Instruction status		
Instruction		Issue	Execute	Write Result
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs[R3]
Add1	Yes	SUB		Mem[32 + Regs[R2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs[F4]	Load2		
Mult2	Yes	DIV		Mem[32 + Regs[R2]]	Mult1		

	Register status								
Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Hardware-Based Speculation

- Execute instructions along predicted execution paths but only commit the results if prediction was correct
- Instruction commit: allowing an instruction to update the register file when instruction is no longer speculative
- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits
  - I.e. updating state or taking an execution

Reorder Buffer

- Reorder buffer – holds the result of instruction between completion and commit
- Four fields:
  - Instruction type: branch/store/register
  - Destination field: register number
  - Value field: output value
  - Ready field: completed execution?
- Modify reservation stations:
  - Operand source is now reorder buffer instead

Reorder Buffer

- Register values and memory values are not written until an instruction commits
- On misprediction:
  - Speculated entries in ROB are cleared
- Exceptions:
  - Not recognized until it is ready to commit

Multiple Issue and Static Scheduling

- To achieve CPI < 1, need to complete multiple instructions per clock

- Solutions:
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - dynamically scheduled superscalar processors

Multiple Issue

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

VLIW Processors

- Package multiple operations into one instruction
- Example VLIW processor:
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references
- Must be enough parallelism in code to fill the available slots

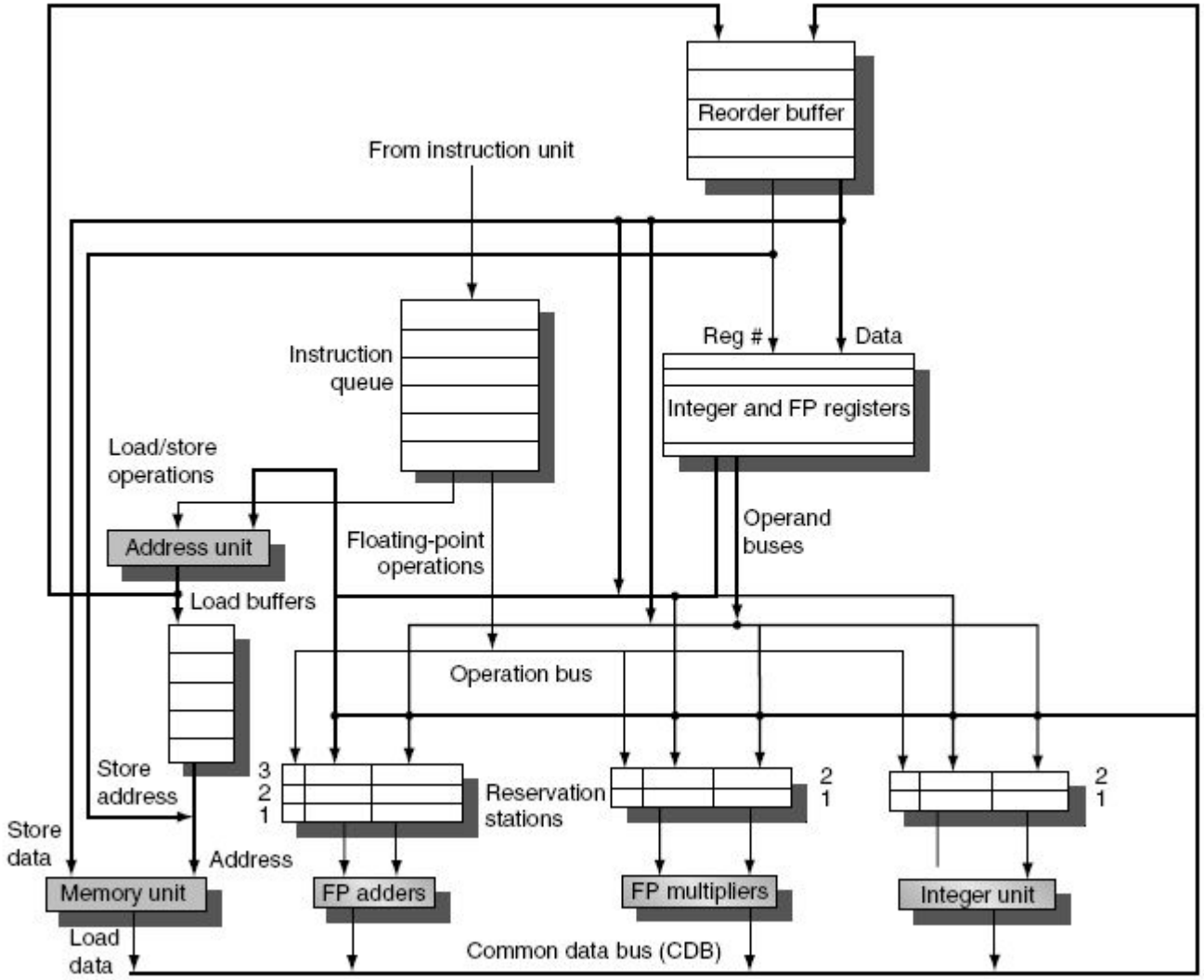
VLIW Processors

- Disadvantages:
  - Statically finding parallelism
  - Code size
  - No hazard detection hardware
  - Binary code compatibility

Dynamic Scheduling, Multiple Issue, and Speculation

- Modern microarchitectures:
  - Dynamic scheduling + multiple issue + speculation
- Two approaches:
  - Assign reservation stations and update pipeline control table in half clock cycles
    - Only supports 2 instructions/clock
  - Design logic to handle any possible dependencies between the instructions
  - Hybrid approaches
- Issue logic can become bottleneck

Overview of Design



Multiple Issue

- Limit the number of instructions of a given class that can be issued in a “bundle”
  - I.e. on FP, one integer, one load, one store
- Examine all the dependencies amoung the instructions in the bundle
- If dependencies exist in bundle, encode them in reservation stations
- Also need multiple completion/commit

Example

```
Loop:  LD R2,0(R1)      ;R2=array element
       DADDIU R2,R2,#1  ;increment R2
       SD R2,0(R1)      ;store result
       DADDIU R1,R1,#8  ;increment pointer
       BNE R2,R3,LOOP   ;branch if not last element
```

Example (No Speculation)

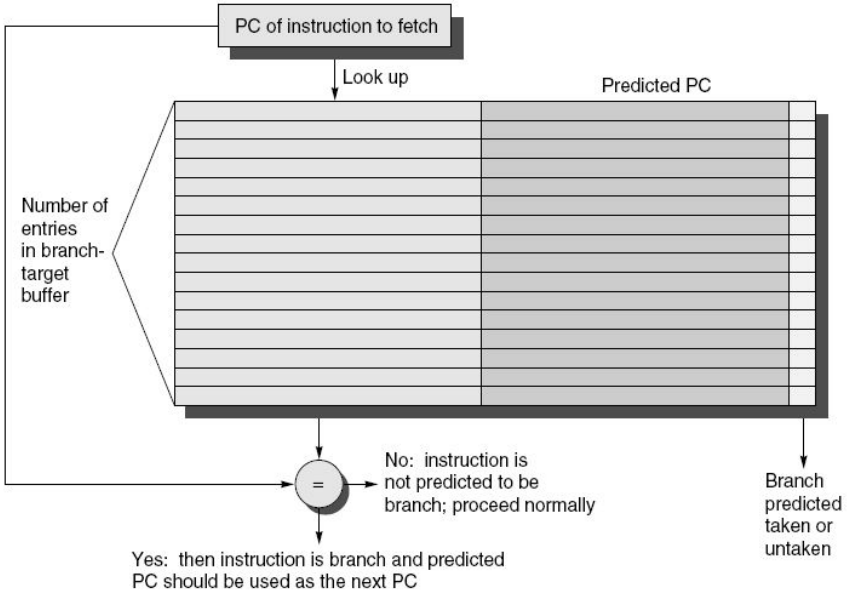
Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#8	8	14		15	Wait for BNE
3	BNE R2,R3,LOOP	9	19			Wait for DADDIU

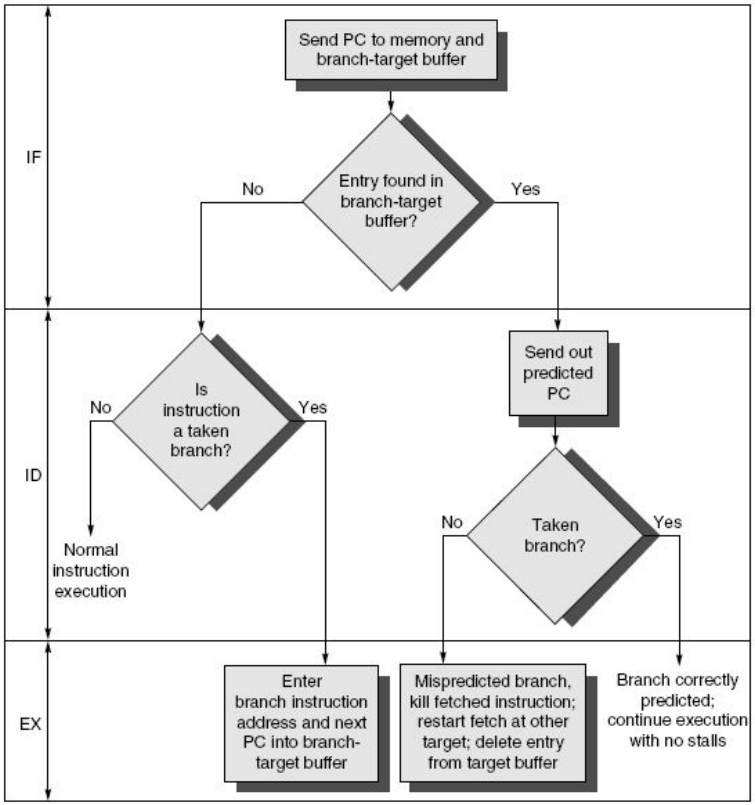
Example

Iteration number	Instructions		Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD	R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU	R2,R2,#1	1	5		6	7	Wait for LW
1	SD	R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU	R1,R1,#8	2	3		4	8	Commit in order
1	BNE	R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD	R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU	R2,R2,#1	4	8		9	10	Wait for LW
2	SD	R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU	R1,R1,#8	5	6		7	11	Commit in order
2	BNE	R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD	R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU	R2,R2,#1	7	11		12	13	Wait for LW
3	SD	R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU	R1,R1,#8	8	9		10	14	Executes earlier
3	BNE	R2,R3,LOOP	9	13			14	Wait for DADDIU

Branch-Target Buffer

- Need high instruction bandwidth!
  - Branch-Target buffers
    - Next PC prediction buffer, indexed by current PC





Branch Folding

- Optimization:
  - Larger branch-target buffer
  - Add target instruction into buffer to deal with longer decoding time required by larger buffer
  - “Branch folding”

Return Address Predictor

- Most unconditional branches come from function returns
- The same procedure can be called from multiple sites
  - Causes the buffer to potentially forget about the return address from previous calls
- Create return address buffer organized as a stack

Integrated Instruction Fetch Unit

- Design monolithic unit that performs:
  - Branch prediction
  - Instruction prefetch
    - Fetch ahead
  - Instruction memory access and buffering
    - Deal with crossing cache lines

Register Renaming

- Register renaming vs. reorder buffers
  - Instead of virtual registers from reservation stations and reorder buffer, create a single register pool
    - Contains visible registers and virtual registers
  - Use hardware-based map to rename registers during issue
  - WAW and WAR hazards are avoided
  - Speculation recovery occurs by copying during commit
  - Still need a ROB-like queue to update table in order
  - Simplifies commit:
    - Record that mapping between architectural register and physical register is no longer speculative
    - Free up physical register used to hold older value
    - In other words: SWAP physical registers on commit
  - Physical register de-allocation is more difficult

Integrated Issue and Renaming

- Combining instruction issue with register renaming:
  - Issue logic pre-reserves enough physical registers for the bundle (fixed number?)
  - Issue logic finds dependencies within bundle, maps registers as necessary
  - Issue logic finds dependencies between current bundle and already in-flight bundles, maps registers as necessary

How Much?

- How much to speculate
  - Mis-speculation degrades performance and power relative to no speculation
    - May cause additional misses (cache, TLB)
  - Prevent speculative code from causing higher costing misses (e.g. L2)
- Speculating through multiple branches
  - Complicates speculation recovery
  - No processor can resolve multiple branches per cycle

Energy Efficiency

- Speculation and energy efficiency
  - Note: speculation is only energy efficient when it significantly improves performance
- Value prediction
  - Uses:
    - Loads that load from a constant pool
    - Instruction that produces a value from a small set of values
  - Not been incorporated into modern processors
  - Similar idea--*address aliasing prediction*--is