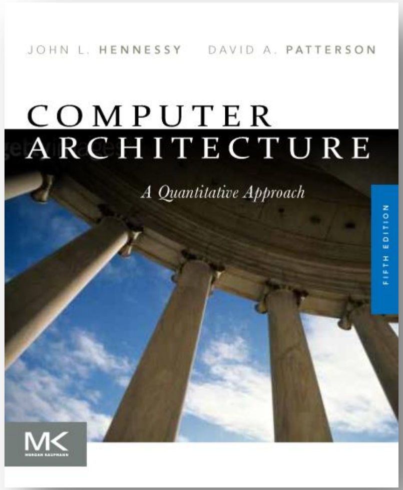


Chapter 3

Instruction-Level Parallelism and Its Exploitation

Contents

Introduction

- Pipelining become universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism (ILP)”
- Two main approaches:
  - Dynamic → hardware-based
    - Used in server and desktop processors
    - Not used as extensively in Parallel Multiprogrammed Microprocessors (PMP)
  - Static approaches → compiler-based
    - Not as successful outside of scientific applications

Review of basic concepts

- Pipelining → each instruction is split up into a sequence of steps – different steps can be executed concurrently by different circuitry.
- A basic pipeline in a RISC processor
  - IF – Instruction Fetch
  - ID – Instruction Decode
  - EX – Instruction Execution
  - MEM – Memory Access
  - WB – Register Write Back
- Two techniques:
  - Superscalar → A superscalar processor executes more than one instruction during a clock cycle.
  - VLIW → very long instruction word – compiler packs multiple independent operations into an instruction

Basic superscalar 5-stage pipeline

Superscalar→ a processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor.

The hardware determines (statically/ dynamically) which one of a block on **n** instructions will be executed next.

	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
i+1	IF	ID	EX	MEM	WB				
i+2		IF	ID	EX	MEM	WB			
i+3		IF	ID	EX	MEM	WB			
i+4			IF	ID	EX	MEM	WB		
i+5			IF	ID	EX	MEM	WB		
i+6				IF	ID	EX	MEM	WB	
i+7				IF	ID	EX	MEM	WB	
i+8					IF	ID	EX	MEM	WB
i+9					IF	ID	EX	MEM	WB

A single-core superscalar processor → SISD

A multi-core superscalar → MIMD.

Hazards→ pipelining could lead to incorrect results.

- Data dependence "true dependence"→ Read after Write hazard (RAW)

  i: sub R1, R2, R3   % sub d,s,t   ➔ d =s-t

  i+1: add R4, R1, R3   % add d,s t   ➔ d = s+t

  Instruction (i+1) reads operand (R1) before instruction (i) writes it.

- Name dependence "anti dependence" → two instructions use the same register or memory location, but there is no data dependency between them.
  - Write after Read hazard (WAR) → Example:

    i: sub R4, R5, R3

    i+1: add R5, R2, R3

    i+2: mul R6, R5, R7

    Instruction (i+1) writes operand (R5) before instruction (i) reads it.
  - Write after Write (WAW) (Output dependence) → Example

    i: sub R6, R5, R3

    i+1: add R6, R2, R3

    i+2: mul R1, R2, R7

    Instruction (i+1) writes operand (R6) before instruction (i) writes it.

More about hazards

- Data hazards ➔ RAW, WAR, WAW.

- Structural hazard ➔ occurs when a part of the processor's hardware is needed by two or more instructions at the same time. Example: a single memory unit that is accessed both in the fetch stage where an instruction is retrieved from memory, and the memory stage where data is written and/or read from memory. They can often be resolved by separating the component into orthogonal units (such as separate caches) or bubbling the pipeline.

- Control hazard (branch hazard) ➔ are due to branches. On many instruction pipeline microarchitectures, the processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline (normally the fetch stage).

Instruction-level parallelism (ILP)

- 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 is s 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

Control dependence

- Every instruction is control dependent on some set of branches and, *in general*, the control dependencies must be preserved to ensure program correctness.
  - 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.
- Example

```
if C1 {
    S1;
};
if C2 {
    S2;
};
```

S1 is control dependent on C1; S2 is control dependent on C2 but not on C1

Properties essential to program correctness

- Preserving exception behavior → any change in instruction order must not change the order in which exceptions are raised. Example:

```
            DADDU    R1, R2, R3
            BEQZ     R1, L1
    L1      LW       R4, 0(R1)
```

  Can we move LW before BEQZ ?

- Preserving data flow → the flow of data between instructions that produce results and consumes them. Example:

```
            DADDU    R1, R2, R3
            BEQZ     R4, L1
            DSUBU    R1, R8, R9
    L1      LW       R5, 0(R2)
            OR       R7, R1, R8
```

  Does OR depend on DADDU and DSUBU?

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

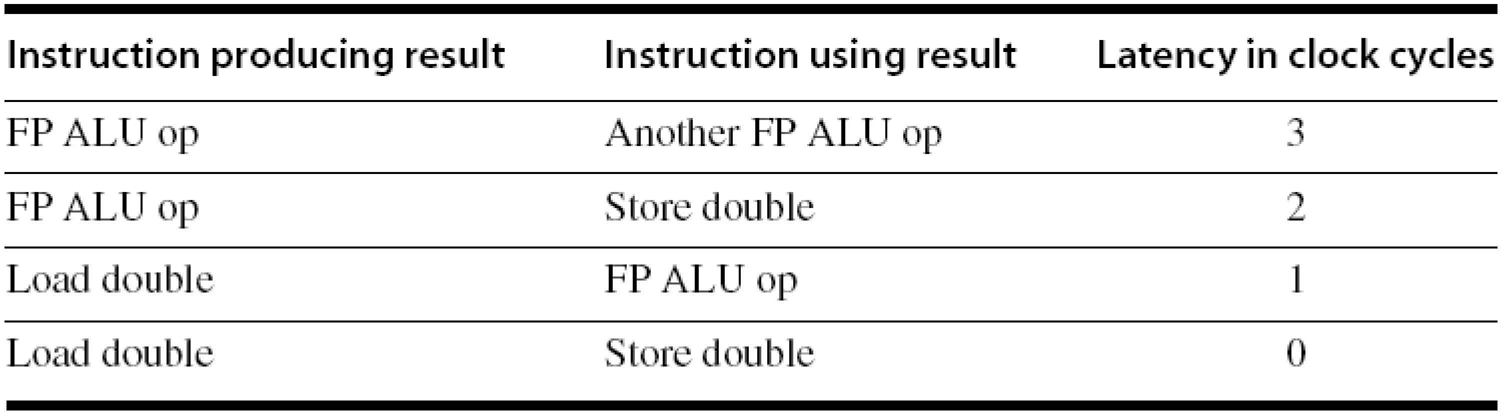
- Loop transformation technique to optimize a program's execution speed:
  1. reduce or eliminate instructions that control the loop, e.g., pointer arithmetic and "end of loop" tests on each iteration
  2. hide latencies, e.g., the delay in reading data from memory
  3. re-write loops as a repeated sequence of similar independent statements → space-time tradeoff
  4. reduce branch penalties;
- Methods

1. pipeline scheduling
2. loop unrolling
3. strip mining
4. branch prediction

1. Pipeline scheduling

- Pipeline stall → delay in execution of an instruction in an instruction pipeline in order to resolve a hazard. The compiler can reorder instructions to reduce the number of pipeline stalls.
- 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;
```



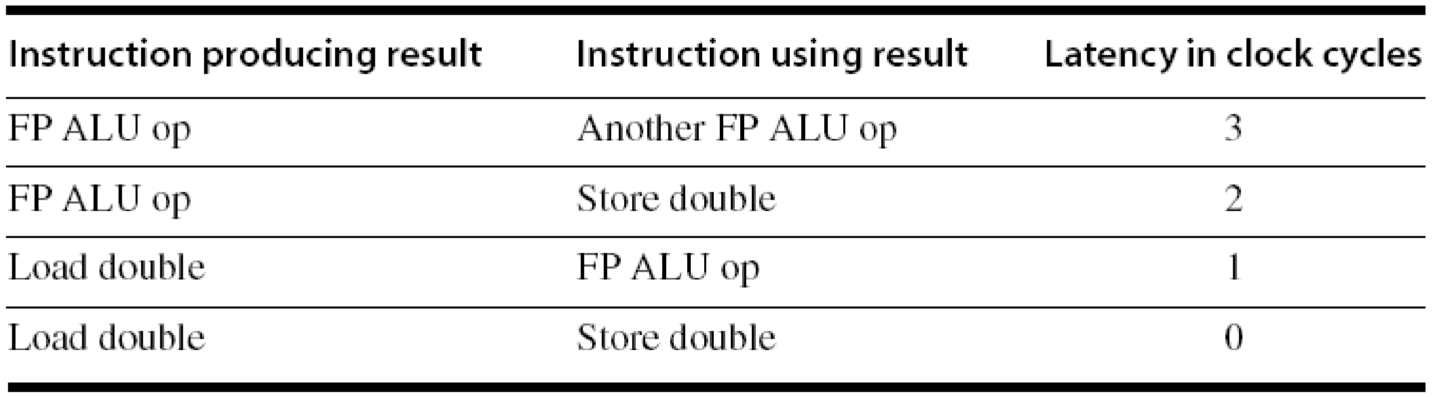
Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Pipeline stalls

Assumptions:

1. S → F2
2. The element of the array with the highest address → R1
3. The element of the array with the lowest address +8 → R2

```
Loop:   L.D       F0,0(R1)        % load array element in F0
        stall                     % next instruction needs the result in F0
        ADD.D     F4,F0,F2        % increment array element
        stall
        stall                     % next instruction needs the result in F4
        S.D       F4,0(R1)        % store array element
        DADDUI    R1,R1,#-8       % decrement pointer to array element
        stall (assume integer load latency is 1) % next instruction needs result in R1
        BNE       R1,R2,Loop      % loop if not the last element
```



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 F     4,8(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

2. Loop unrolling

- Given the same code

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

- Assume # elements of the array with starting address in R1 is divisible by 4
- Unroll by a factor of 4
- Eliminate unnecessary instructions.

Unrolled loop

```
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
```

- Live registers: F0-1, F2-3, F4-5, F6-7, F8-9,F10-11, F12-13, F14-15, and F16-15; also R1 and R2
- In the original code: only F0-1, F2-3, and F4-5; also R1 and R2

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
```

Pipeline scheduling reduces the number of stalls.

1. The L.D instruction requires only one cycle so when ADD.D are issued F4, F8, F12 , and F16 are already loaded.

2. The ADD.D requires only two cycles so that two S.D can proceed immediately

3. The array pointer is updated after the first two S.D so the loop control can proceed immediately after the last two S.D

Loop unrolling & scheduling summary

- Use different registers to avoid unnecessary constraints.
- Adjust the loop termination and iteration code.
- Find if the loop iterations are independent except the loop maintenance code ➔ if so unroll the loop
- Analyze memory addresses to determine if the load and store from different iterations are independent ➔if so interchange load and stores in the unrolled loop
- Schedule the code while ensuring correctness.
- Limitations of loop unrolling
  - Decrease of the amount of overhead with each roll
  - Growth of the code size
  - Register pressure (shortage of registers) → scheduling to increase ILP increases the number of live values thus, the number of registers

More compiler-based loop transformations

- fission/distribution →break a loop into multiple loops over the same index range but each taking only a part of the loop's body. Can improve locality of reference, both of the data being accessed in the loop and the code in the loop's body.
- fusion/combining → when two adjacent loops would iterate the same number of times their bodies can be combined as long as they make no reference to each other's data.
- interchange/permutation → exchange inner loops with outer loops. When the loop variables index into an array, such a transformation can improve locality of reference.
- inversion → changes a standard *while* loop into a *do/while* (a.k.a. *repeat/until*) loop wrapped in an *if* conditional, reducing the number of jumps by two for cases where the loop is executed. Doing so duplicates the condition check (increasing the size of the code) but is more efficient because jumps usually cause a pipeline stall.
- reversal → reverses the order in which values are assigned to the index variable. This is a subtle optimization which can help eliminate dependencies and thus enable other optimizations.
- scheduling → divide a loop into multiple parts that may be run concurrently on multiple processors.
- skewing → take a nested loop iterating over a multidimensional array, where each iteration of the inner loop depends on previous iterations, and rearranges its array accesses so that the only dependencies are between iterations of the outer loop.

3. Strip mining

- The block size of the outer block loops is determined by some characteristic of the target machine, such as the vector register length or the cache memory size.
  - 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”

4. Branch prediction

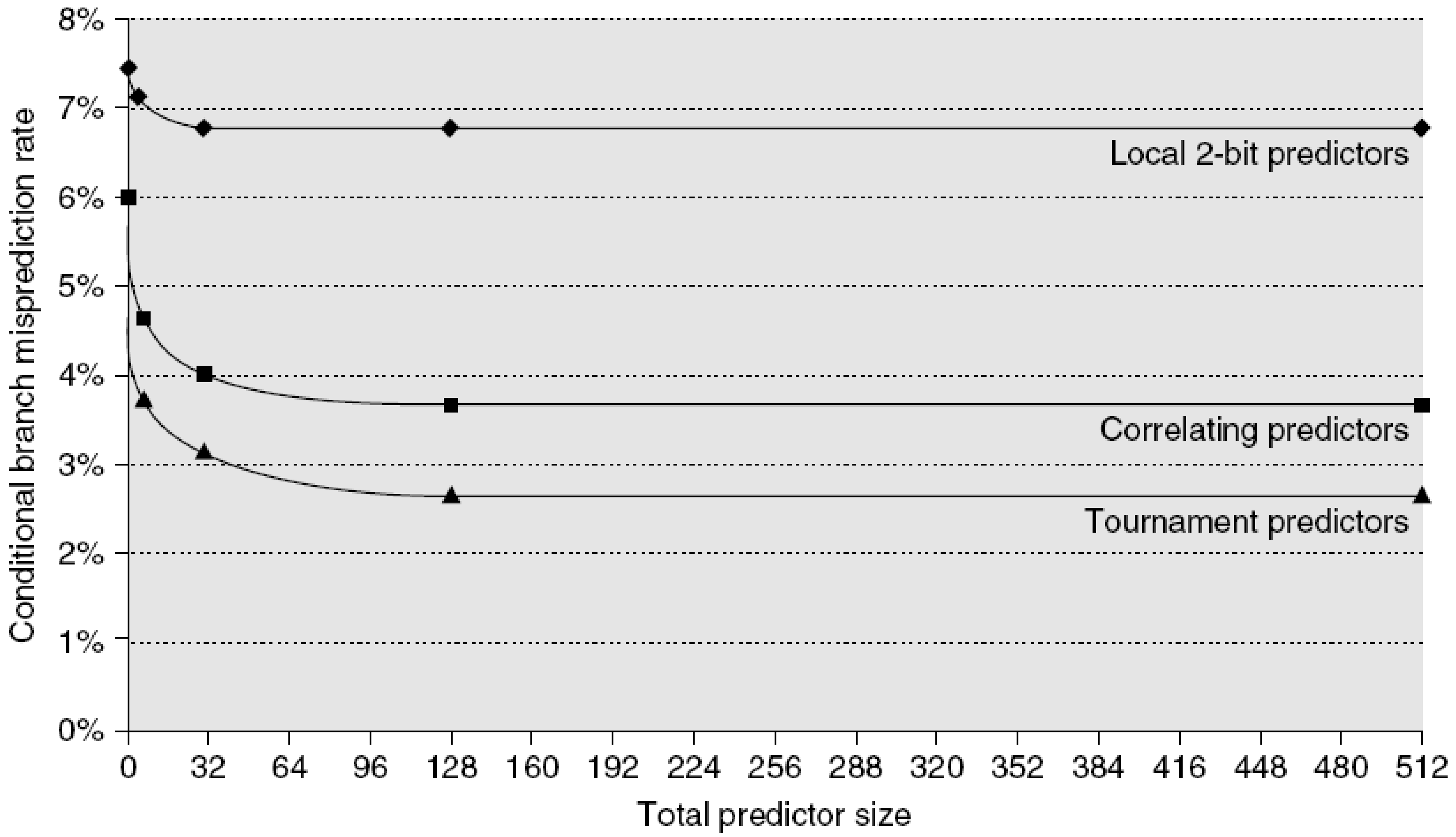
- Branch prediction → guess whether a conditional jump will be taken or not.
- Goal → improve the flow in the instruction pipeline.
- Speculative execution → the branch that is guessed to be the most likely is then fetched and speculatively executed.
- Penalty → if it is later detected that the guess was wrong then the speculatively executed or partially executed instructions are discarded and the pipeline starts over with the correct branch, incurring a delay.
- How? → the branch predictor keeps records of whether branches are taken or not taken. When it encounters a conditional jump that has been seen several times before then it can base the prediction on the history. The branch predictor may, for example, recognize that the conditional jump is taken more often than not, or that it is taken every second time

- Note: not to be confused with branch target prediction → guess the target of a taken conditional or unconditional jump before it is computed by decoding and executing the instruction itself. Both are often combined into the same circuitry.

Predictors

- Basic 2-bit predictor:
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding *n* branches
- Local predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the last *n* occurrences of this branch
- Tournament predictor:
  - Combine correlating predictor with local predictor

Branch prediction performance



Dynamic scheduling

- Rearrange order of instructions to reduce stalls while maintaining data flow
- 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:

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

antidependence WAR - F8

antidependence WAW – F6

+ name dependence with F6

Register renaming

- Example: add two temporary registers S and T

```
i:      DIV.D   F0,F2,F4
i+1:    ADD.D   S,F0,F8      (instead of  ADD.D   F6,F0,F8)
i+2     S.D     S,0(R1)      (instead of  S.D     F6,0(R1)
i+3     SUB.D   T,F10,F14    (instead of  SUB.D   F8,F10,F14)
i+4     MUL.D   F6,F10,T     (instead of  MUL.D   F6,F10,F8)
```

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

Register renaming

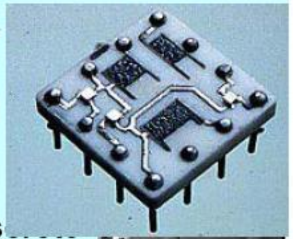
- Register renaming is provided by reservation stations (RS)
  - Contains:
    - The instruction
    - Buffered operand values (when available)
    - Reservation station number of instruction providing the operand values
  - RS fetches and buffers an operand as soon as it becomes available (not necessarily 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

- Goal → high performance without special compilers when the hardware has only a small number of floating point (FP) registers.
- Designed in 1966 for IBM 360/91 with only 4 FP registers.
- Used in modern processors: Pentium 2/3/4, Power PC 604, Neehalem…..
- Additional hardware needed
  - Load and store buffers → contain data and addresses, act like reservation station.
  - Reservation stations → feed data to floating point arithmetic units.

IBM360/91

- CPU cycle time: 60 nanoseconds
- memory cycle time (to fetch and store eight bytes in parallel): 780ns
- Standard memory capacity: 2,097,152B interleaved 16 ways (magnetic cores)
- Up to 6,291,496 bytes of main storage
- Up to 16.6-million additions/second
- Ca.120K gates, ECL
- Solid Logic Technology (SLT), an IBM invention which encapsulated 5-6 transistors into a small module--a transition technology between dis transistors and the IC
- About 12 were made

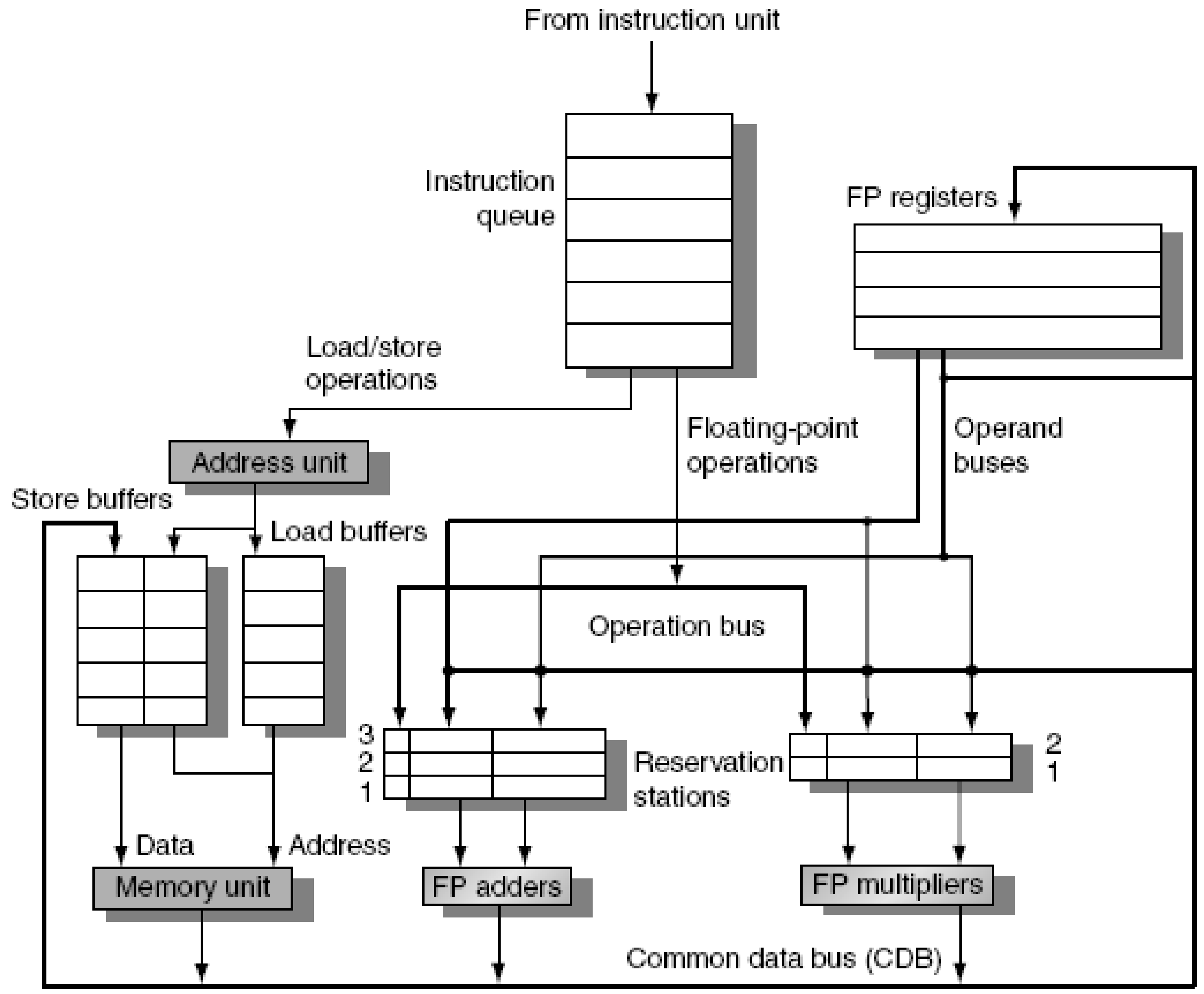


NASA's Space Flight Center in Greenbelt, Md, January 1968

NASA Center for Computational Sciences

See:
Some Reflections on Computer Engineering: 30 Years after the IBM System 360 Model 91
Michael J. Flynn
ftp://arith.stanford.edu/tr/micro30.ps.Z

Source:
http://www.columbia.edu/acis/history/36091.html


From instruction unit
Instruction queue
FP registers
Load/store operations
Floating-point operations
Operand buses
Address unit
Store buffers
Load buffers
Operation bus
3
2
1
Reservation stations
2
1
Data
Address
Memory unit
FP adders
FP multipliers
Common data bus (CDB)

Instruction execution 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
  - Broadcast result on CDB into reservation stations and store buffers
    - (Stores must wait until address and value are received)

Example

Instruction		Instruction status		
		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			

Notations

Op—Operation to perform in the unit (e.g., + or –)

Qj, Qk—Reservation stations producing source registers (value to be written)

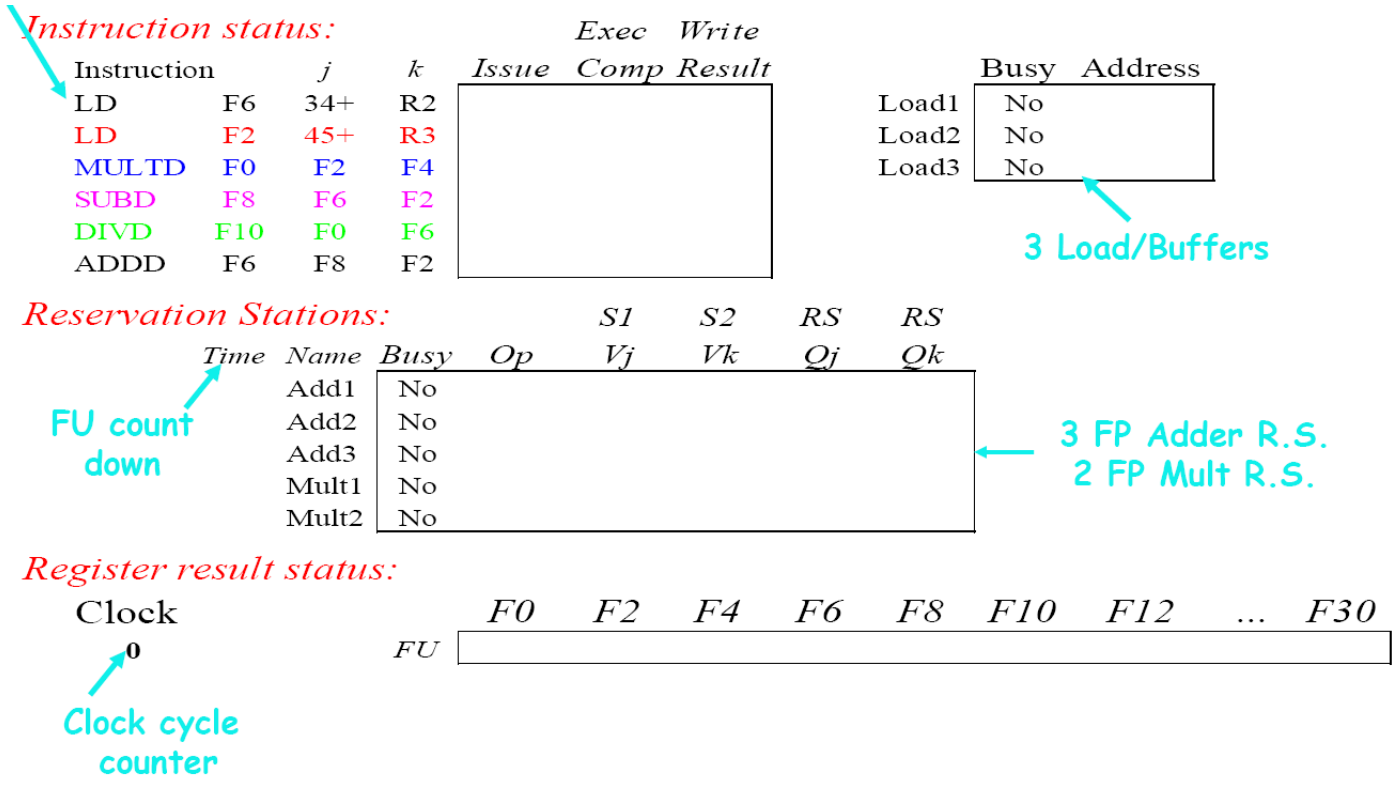
Vj, Vk—Value of Source operands

Rj, Rk—Flags indicating when Vj, Vk are ready

Busy—Indicates reservation station and FU is busy

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Example of Tomasulo algorithm



Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
0	FU									

3 Load/Buffers
FU count down
3 FP Adder R.S.
2 FP Mult R.S.
Clock cycle counter

- 3 – load buffers (Load1, Load 2, Load 3)
- 5 - reservation stations (Add1, Add2, Add3, Mult 1, Mult 2).
- 16 pairs of floating point registers F0-F1, F2-F3,…..F30-31.
- Clock cycles: addition → 2; multiplication → 10; division →40;

Clock cycle 1

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations:

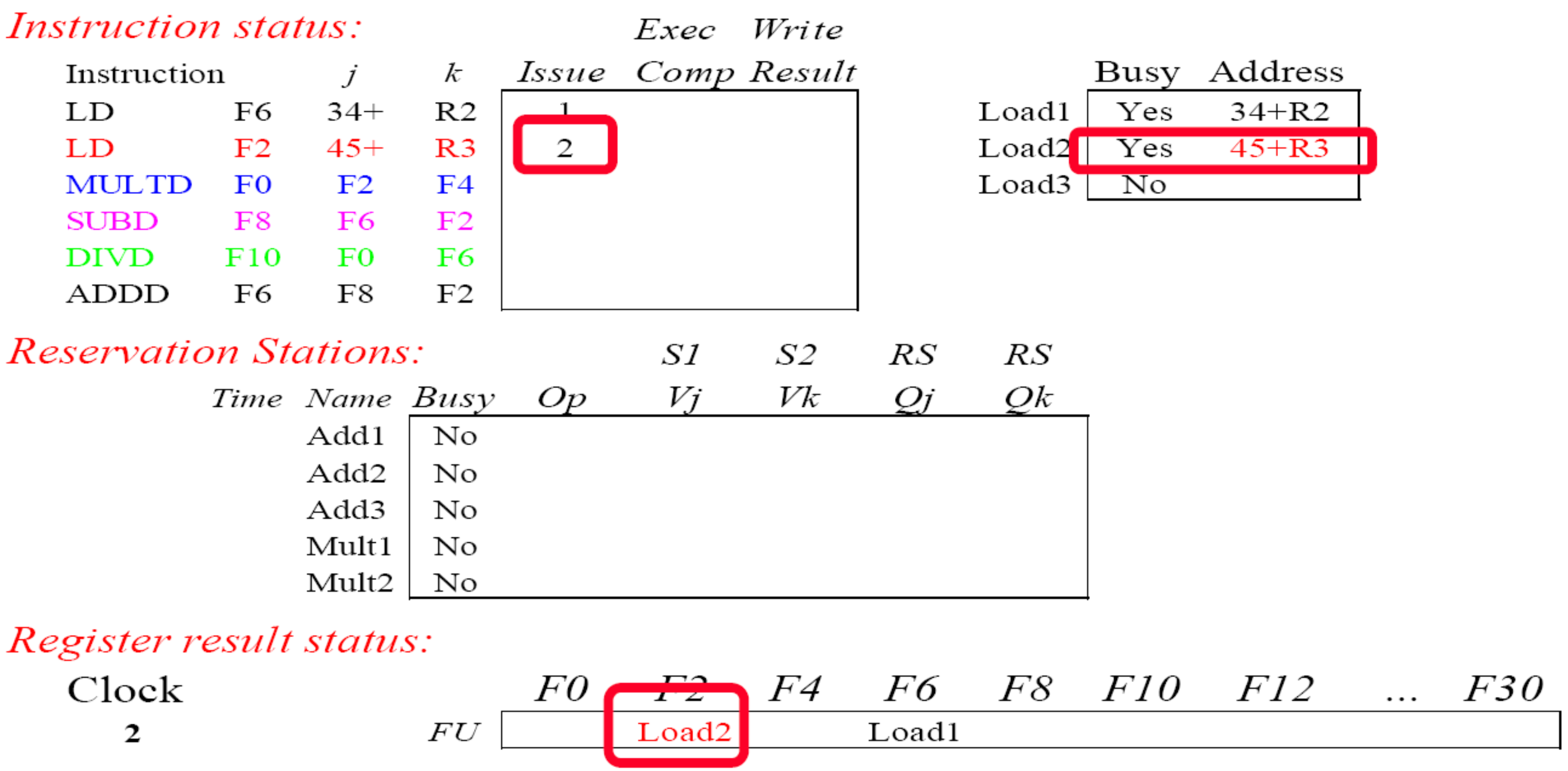
				S1	S2	RS	RS
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
1	*FU*				Load1					

- A load from memory location 34+(R2) is issued; data will be stored later in the first load buffer (Load1)

Clock cycle 2



Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

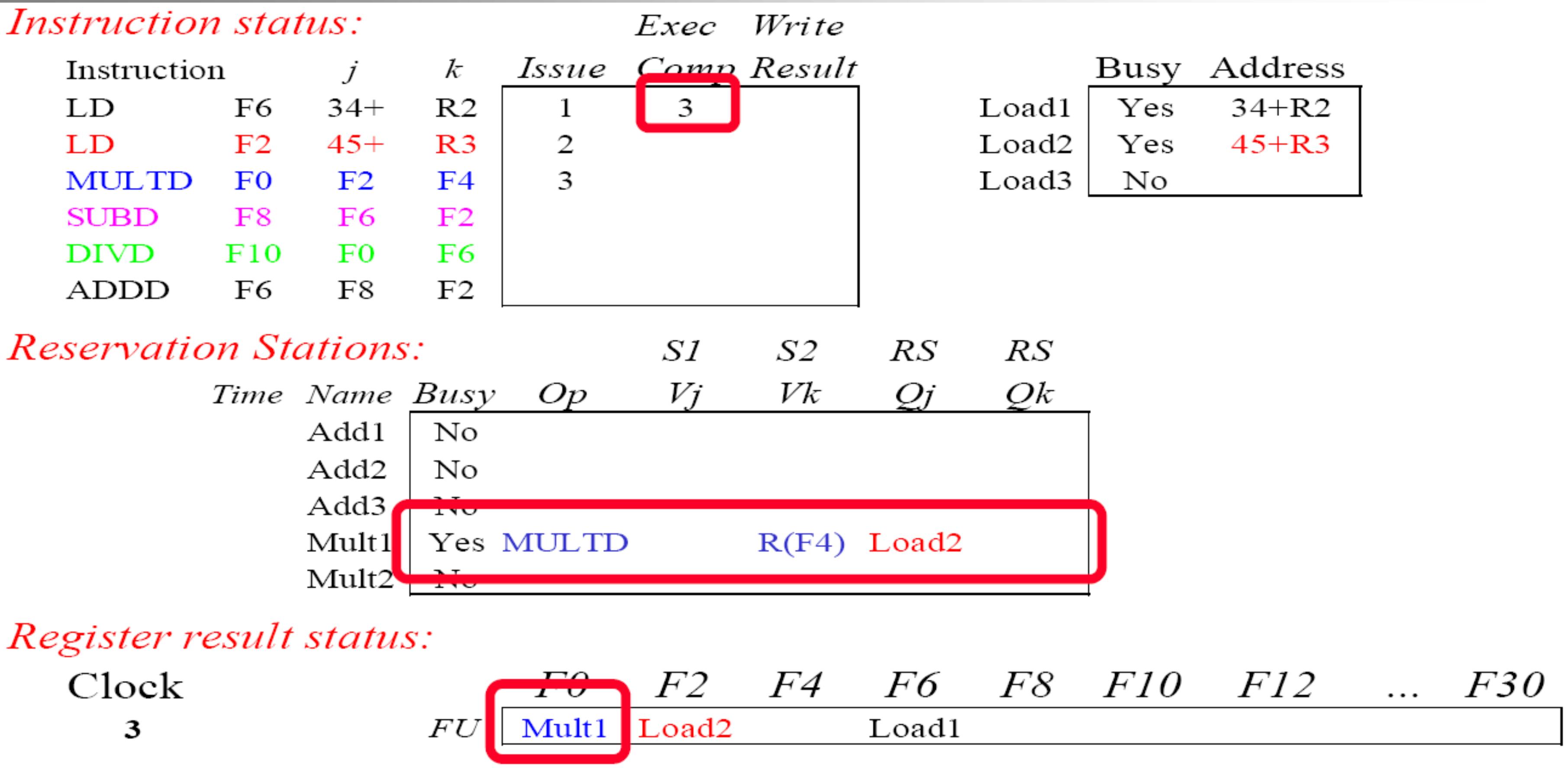
				S1	S2	RS	RS
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU		Load2		Load1					

- The Second load from memory loaction 45+(R3) is issued; data will be stored in load buffer 2 (Load2).
- Multiple loads can be outstanding.

Clock cycle 3



Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

				S1	*S2*	*RS*	*RS*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
3	*FU*	Mult1	Load2		Load1					

- First load completed → SUBD instruction is waiting for the data.
- MULTD → issued
- Register names are removed/renamed in Reservation Stations

Clock cycle 4

Instruction status:

Instruction	j	k		Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

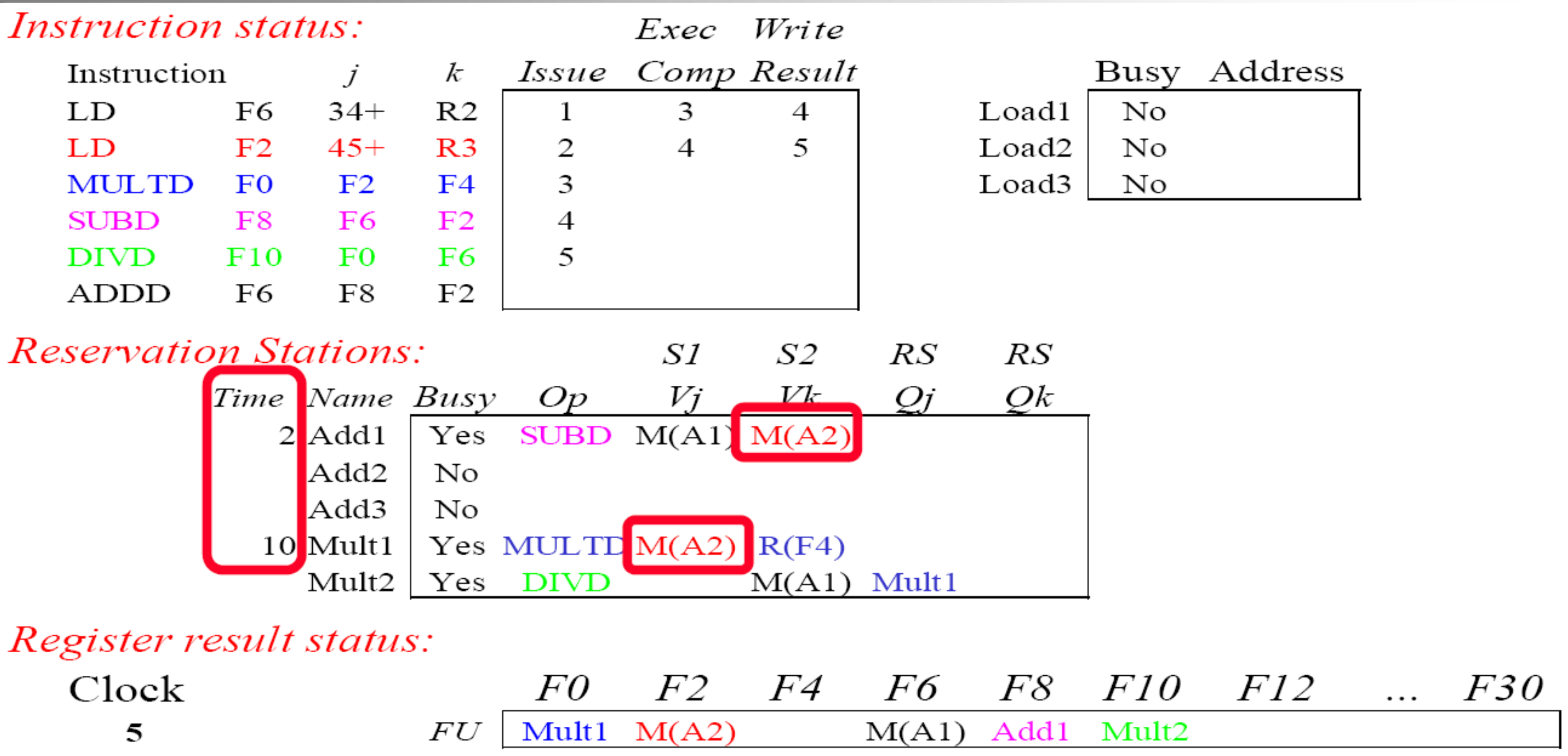
Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	Yes	SUBD	M(A1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2		M(A1)	Add1				

- Load2 is completed; MULTD waits for the results of it.
- The results of Load 1 are available for the SUBD instruction.

Clock cycle 5



Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

				S1	S2	RS	RS
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU	Mult1	M(A2)		M(A1)	Add1	Mult2			

- The result of Load2 available for MULTD executed by Mult1 and SUBD executed by Add1. Both can now proceed as they have both operands.
- Mult2 executes DIVD and cannot proceed yet as it waits for the results of Add1.

Clock cycle 6

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

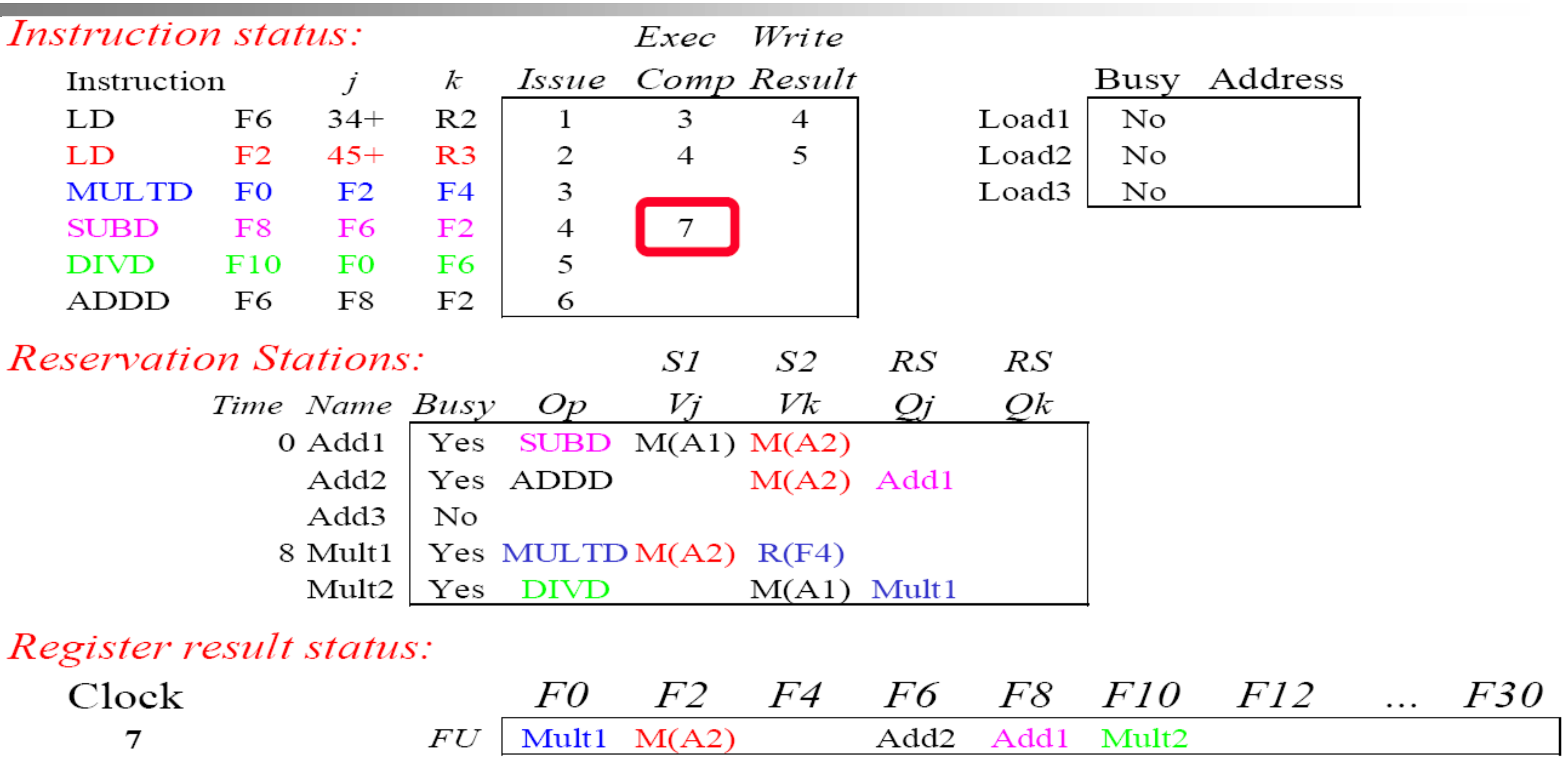
				S1	S2	RS	RS
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

- Issue ADDD

Clock cycle 7



Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	$\boxed{7}$	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

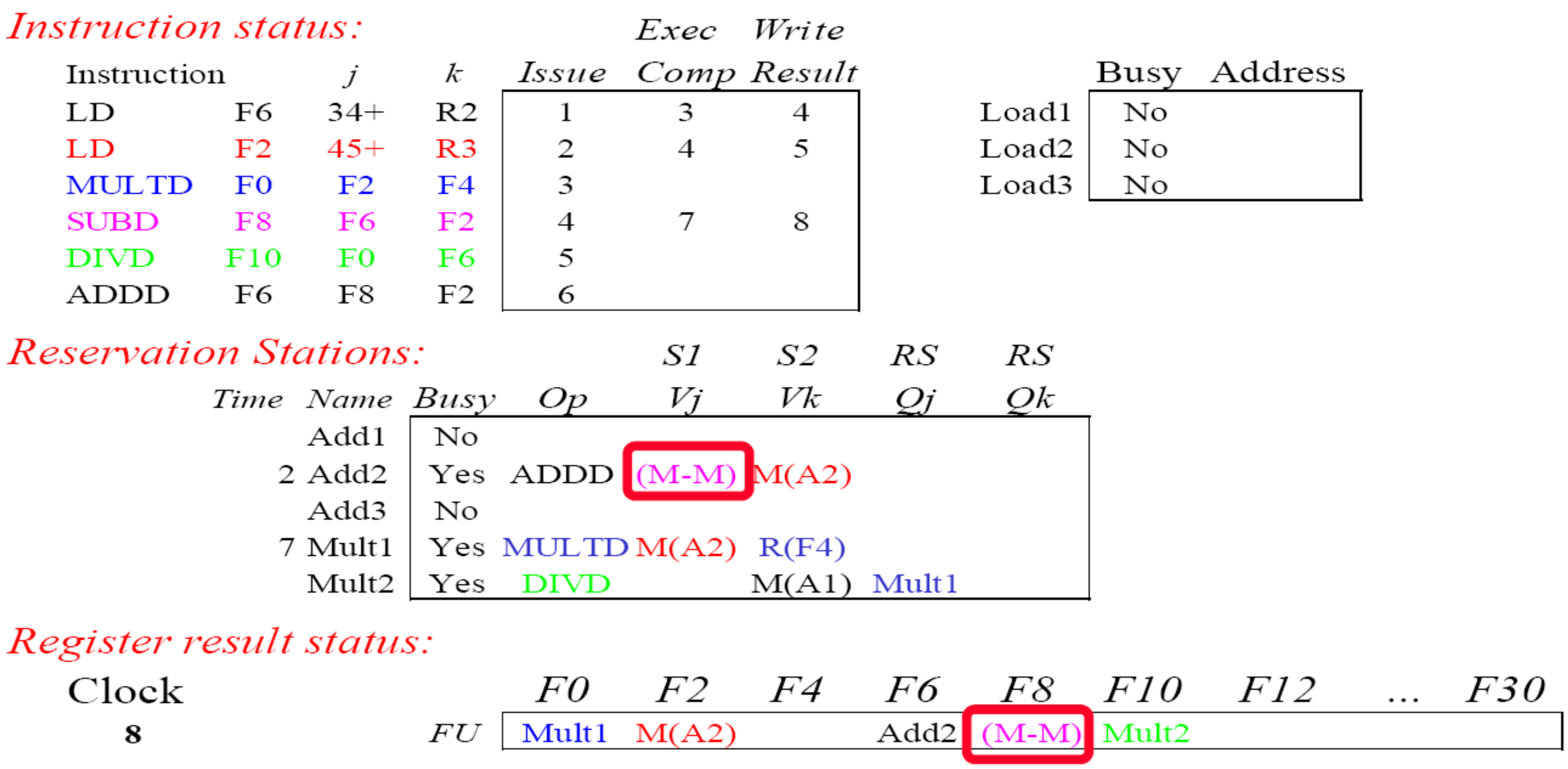
				S1	S2	RS	RS
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
7	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

- The results of the SUBD produced by Add1 will be available in the next cycle.
- ADDD instruction executed by Add2 waits for them.

Clock cycle 8



Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
8	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

- The results of SUBD are deposited by the Add1 in F8-F9

Clock cycle 9

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
9	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Clock cycle 10

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

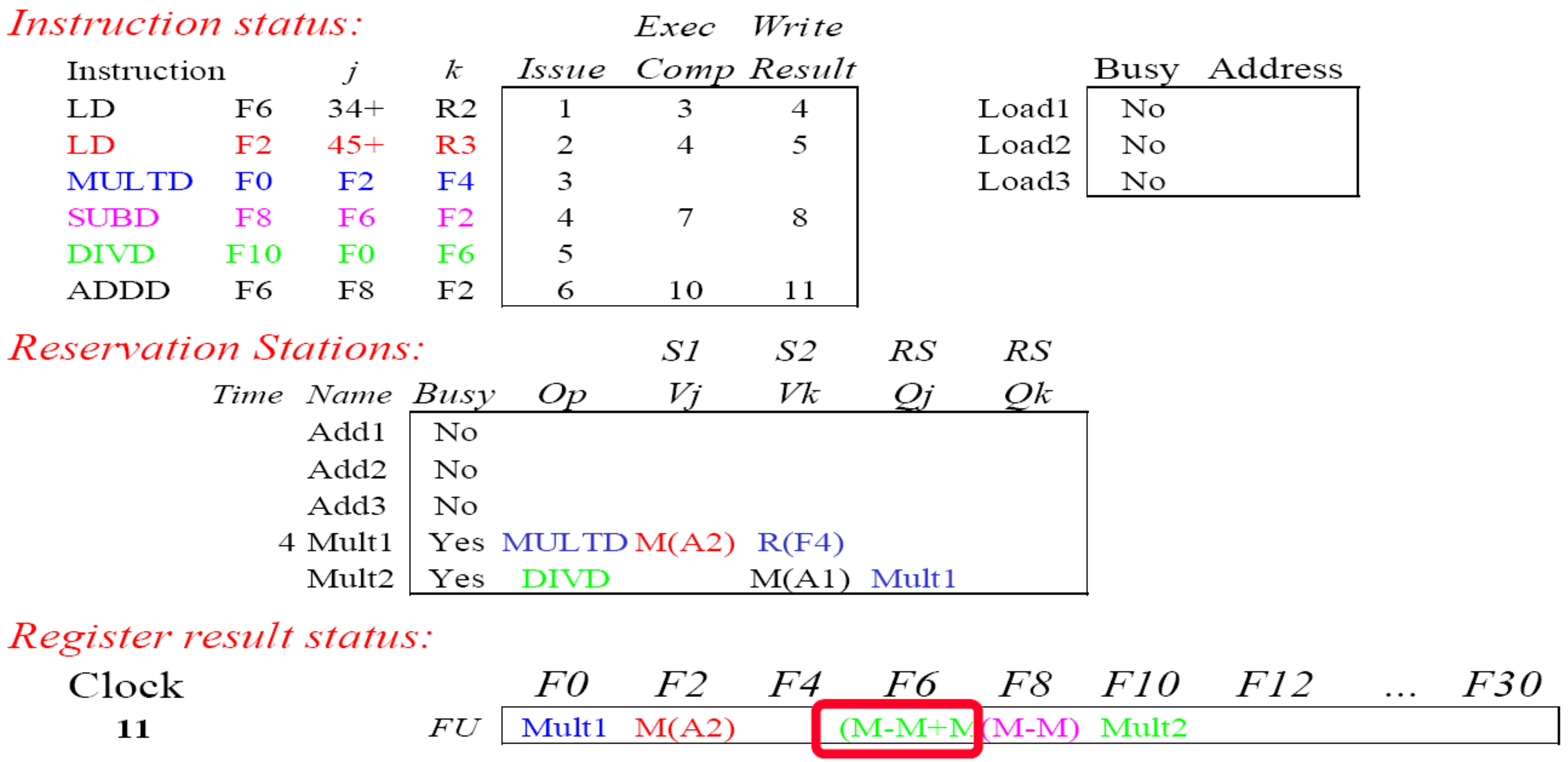
Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
10	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

- ADDD executed by Add2 completes, it needed 2 cycles.
- There are 5 more cycles for MULTD executed by Mult1.

Clock cycle 11



Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

				S1	S2	RS	RS
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
11	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- Only MULTD and DIVD instructions did not complete. DIVD is waiting for the result of MULTD before moving to the execute stage.

Clock cycle 12

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
12	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Clock cycle 13

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
13	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Clock cycle 14

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

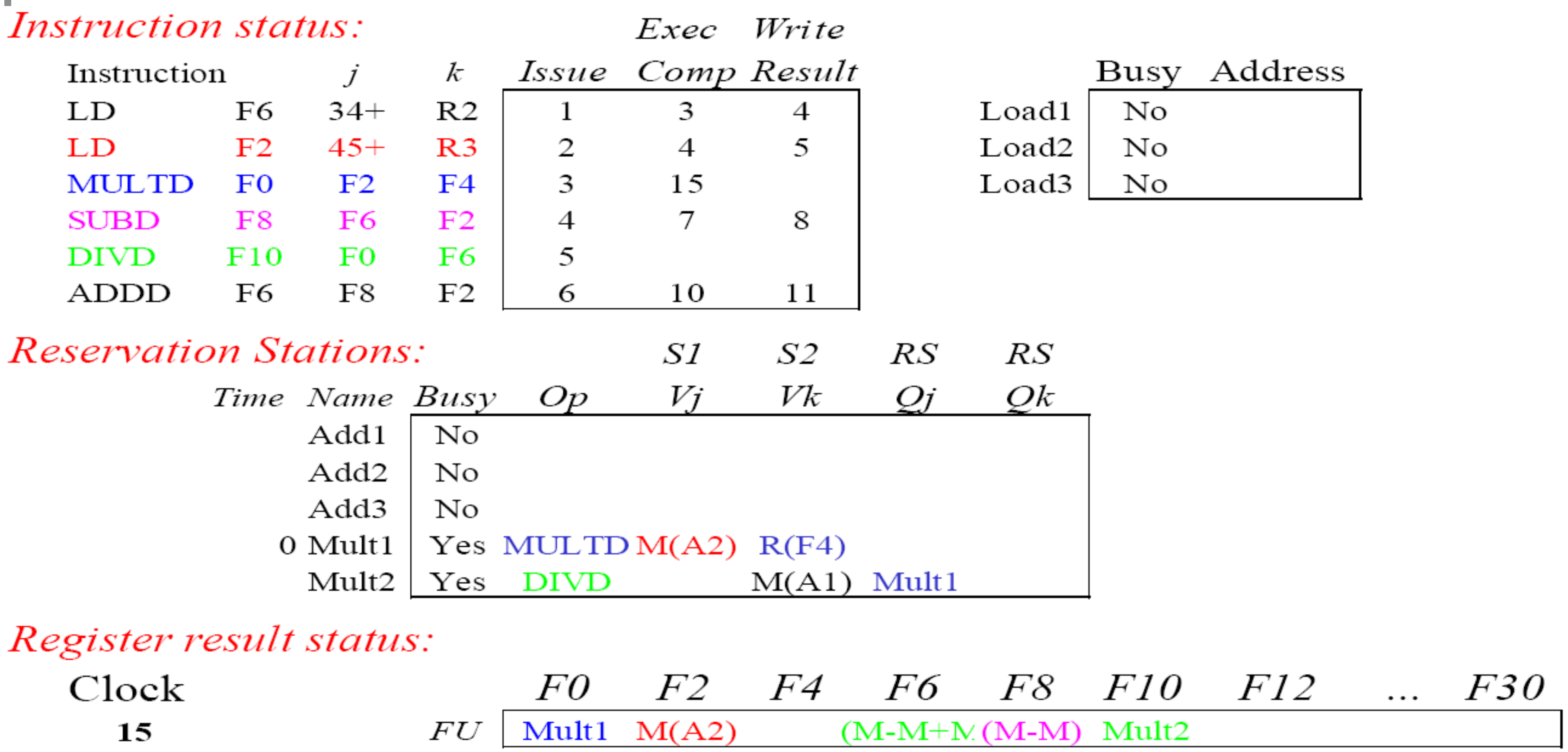
Reservation Stations:

				S1	S2	RS	RS
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
14	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Clock cycle 15



Instruction status:

Instruction	j	k		Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
15	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- MULTS instruction executed by Mult1 unit completed execution.
- DIVD in instruction executed by the Mult2 unit is waiting for it.

Clock cycle 16

Instruction status:

Instruction	*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	5
MULTD F0	F2	F4	3	15	16
SUBD F8	F6	F2	4	7	8
DIVD F10	F0	F6	5		
ADDD F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
16	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

Clock cycle 55

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
1	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
55	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- DIVD will finish execution in cycle 56 and the result will be in F6-F7 in cycle 57.

Hardware-based speculation

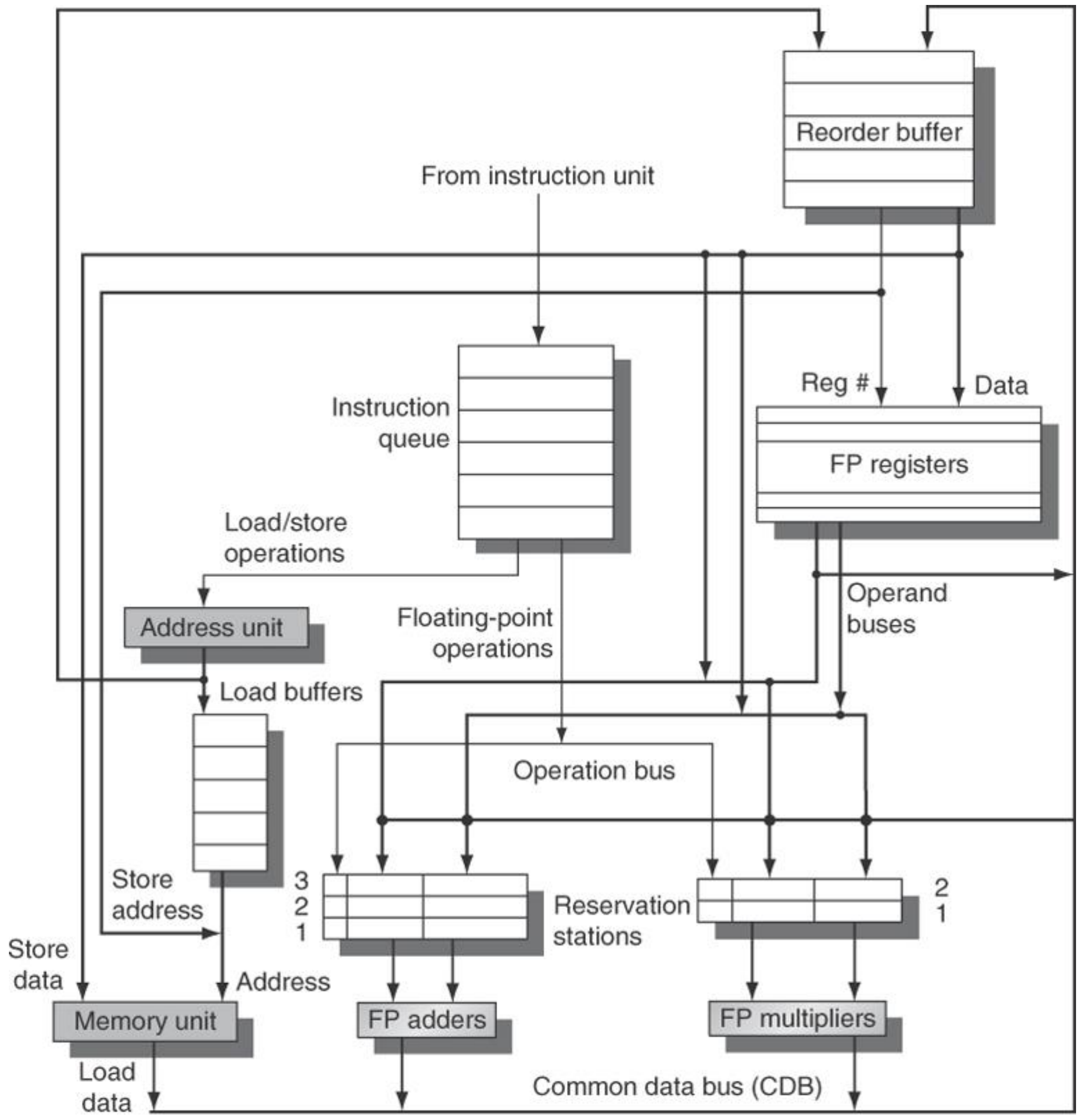
- Goal → overcome control dependency by speculating.
- Allow instructions to execute out of order but force them to commit to avoid: (i) updating the state or (ii) taking an exception
- Instruction commit → allow an instruction to update the register file when instruction is no longer speculative
- Key ideas:
    1. Dynamic branch prediction.
    2. Execute instructions along predicted execution paths, but only commit the results if prediction was correct.
    3. Dynamic scheduling to deal with different combination of basic blocks

How speculative execution is done

- Need additional hardware to prevent any irrevocable action until an instruction commits.
  - Reorder buffer (ROB)
  - Modify functional units – operand source is ROB rather than functional units
- 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

Extended floating point unit

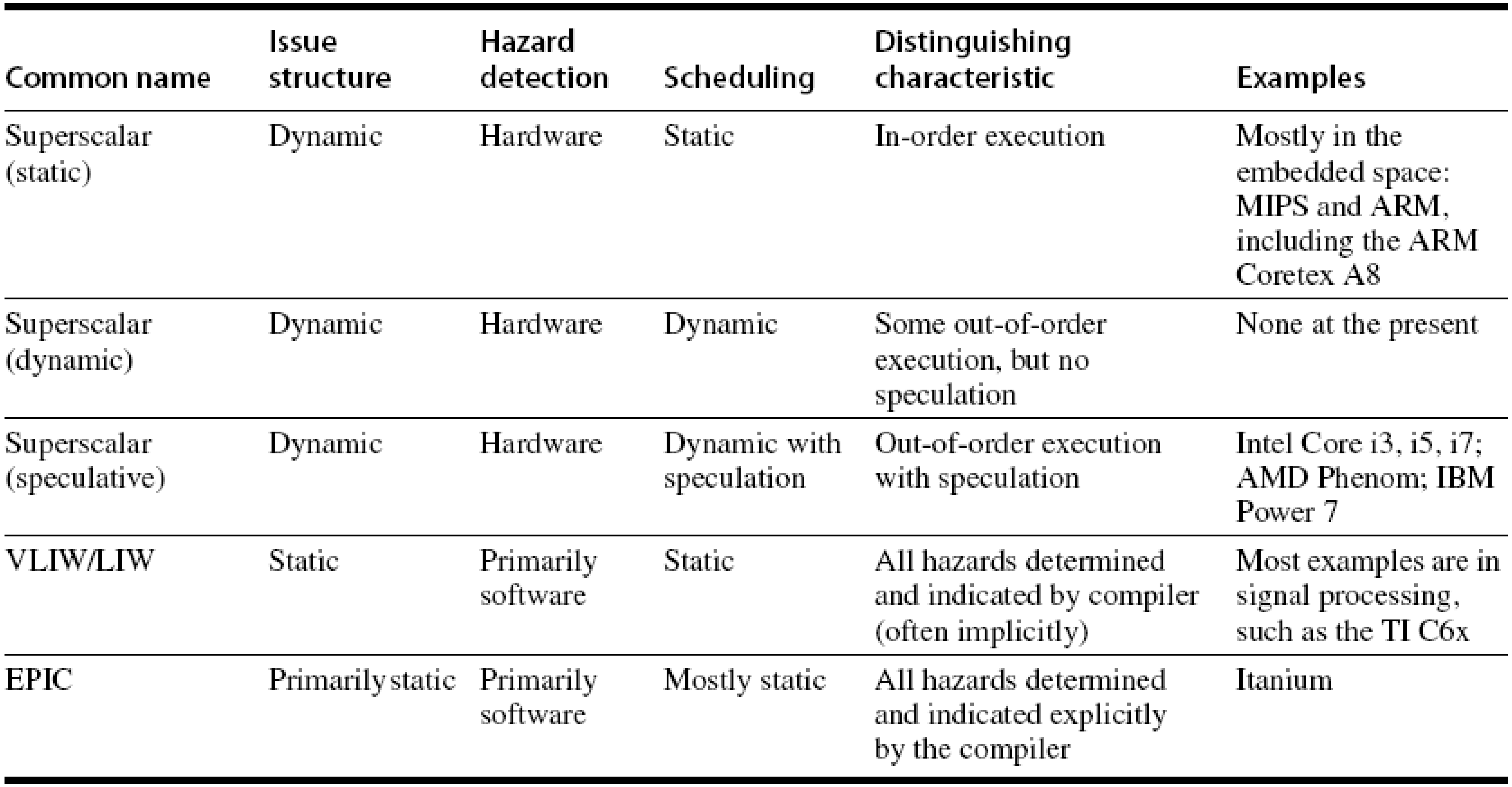
- FP using Tomasulo's algorithm extended to handle speculation.
- Reorder buffer now holds the result of instruction between completion and commit. Has 4 fields
  - Instruction type: branch/store/register
  - Destination field: register number
  - Value field: output value
  - Ready field: completed execution?
- Operand source is now reorder buffer instead of functional unit



Multiple issue and static scheduling

- To achieve CPI < 1➔ complete multiple instructions per clock cycle
- Three flavors of multiple issue processors
  1. Statically scheduled superscalar processors
     a. Issue a varying number of instructions per clock cycle
     b. Use in-order execution
  2. VLIW (very long instruction word) processors
     a. Issue a fixed number of instructions as one large instruction
     b. Instructions are statically scheduled by the compiler
  3. Dynamically scheduled superscalar processors
     a. Issue a varying number of instructions per clock cycle
     b. Use out-of-order execution

Multiple issue processors



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
- Must be enough parallelism in code to fill the available slots,
- Disadvantages:
  - Statically finding parallelism
  - Code size
  - No hazard detection hardware
  - Binary code compatibility

Example

- Unroll the loop for x[i]= x[i] +s to eliminate any stalls. Ignore delayed branches.
- The code we had before shown on the right ➔
- Package in one VLIW instruction :
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references.

```
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
```

Example

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

Figure 3.16 VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes 9 cycles assuming no branch delay; normally the branch delay would also need to be scheduled. The issue rate is 23 operations in 9 clock cycles, or 2.5 operations per cycle. The efficiency, the percentage of available slots that contained an operation, is about 60%. To achieve this issue rate requires a larger number of registers than MIPS would normally use in this loop. The VLIW code sequence above requires at least eight FP registers, while the same code sequence for the base MIPS processor can use as few as two FP registers or as many as five when unrolled and scheduled.

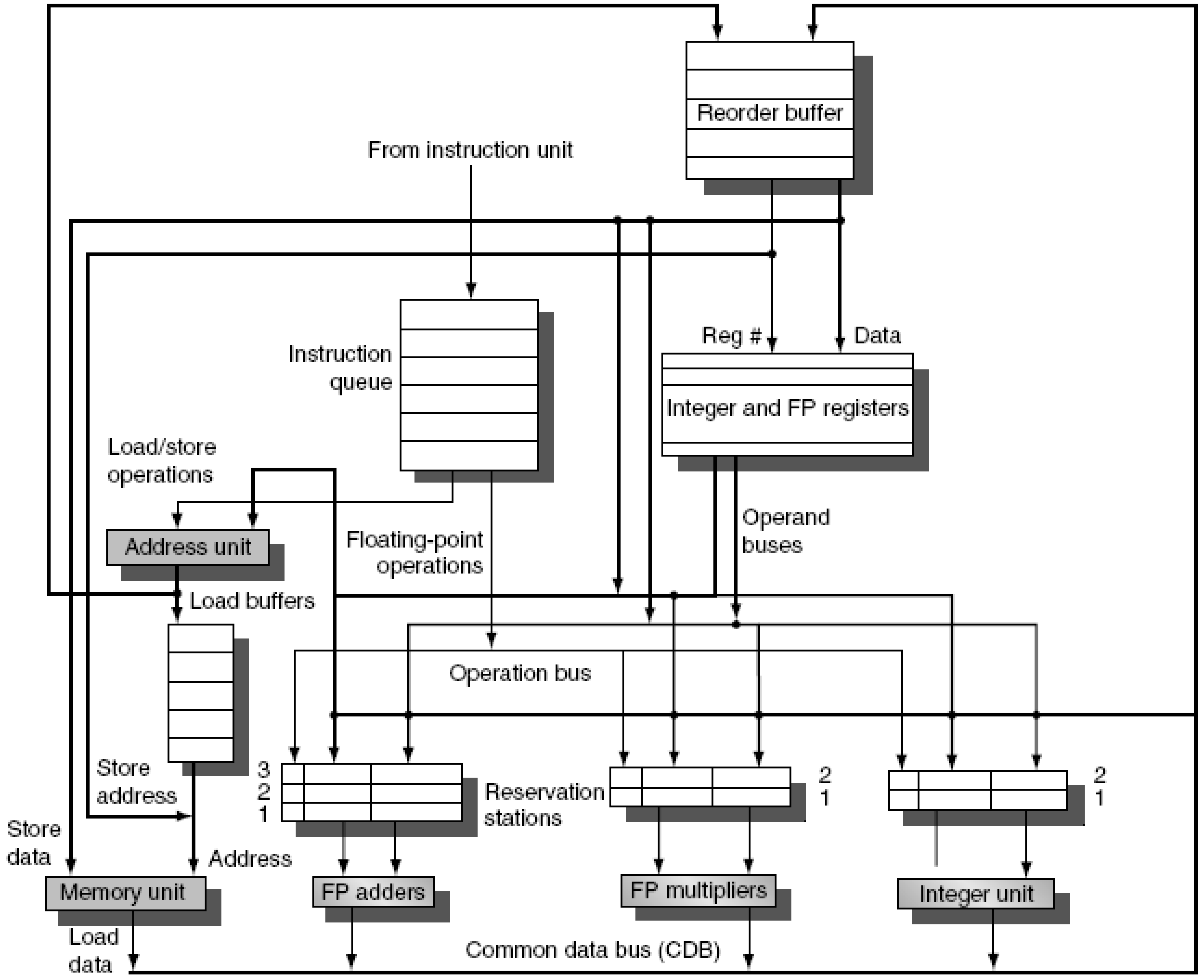
Dynamic scheduling, multiple issue, speculation

- 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

Multiple issue processor with speculation

- The organization should allow simultaneous execution for all issues in one clock cycle of one of the following operations:
  - FP multiplication
  - FP addition
  - Integer operations
  - Load/Store
- Several datapaths must be widened to support multiple issues.
- The instruction issue logic will be fairly complex.

Multiple issue processor with speculation



Basic strategy for updating the issue logic

- Assign a reservation station and a reorder buffer for every instruction that may be issued in the next bundle.
- To pre-allocate reservation stations limit the number of instructions of a given class that can be issued in a “bundle”
  - I.e. one FP, one integer, one load, one store
- Examine all the dependencies among the instructions in the bundle
- If dependencies exist in bundle, use the assigned ROB number to update the reservation table for dependent instructions. Otherwise, use the existing reservations table entries for the issuing instruction.

- Also need multiple completion/commit

Example

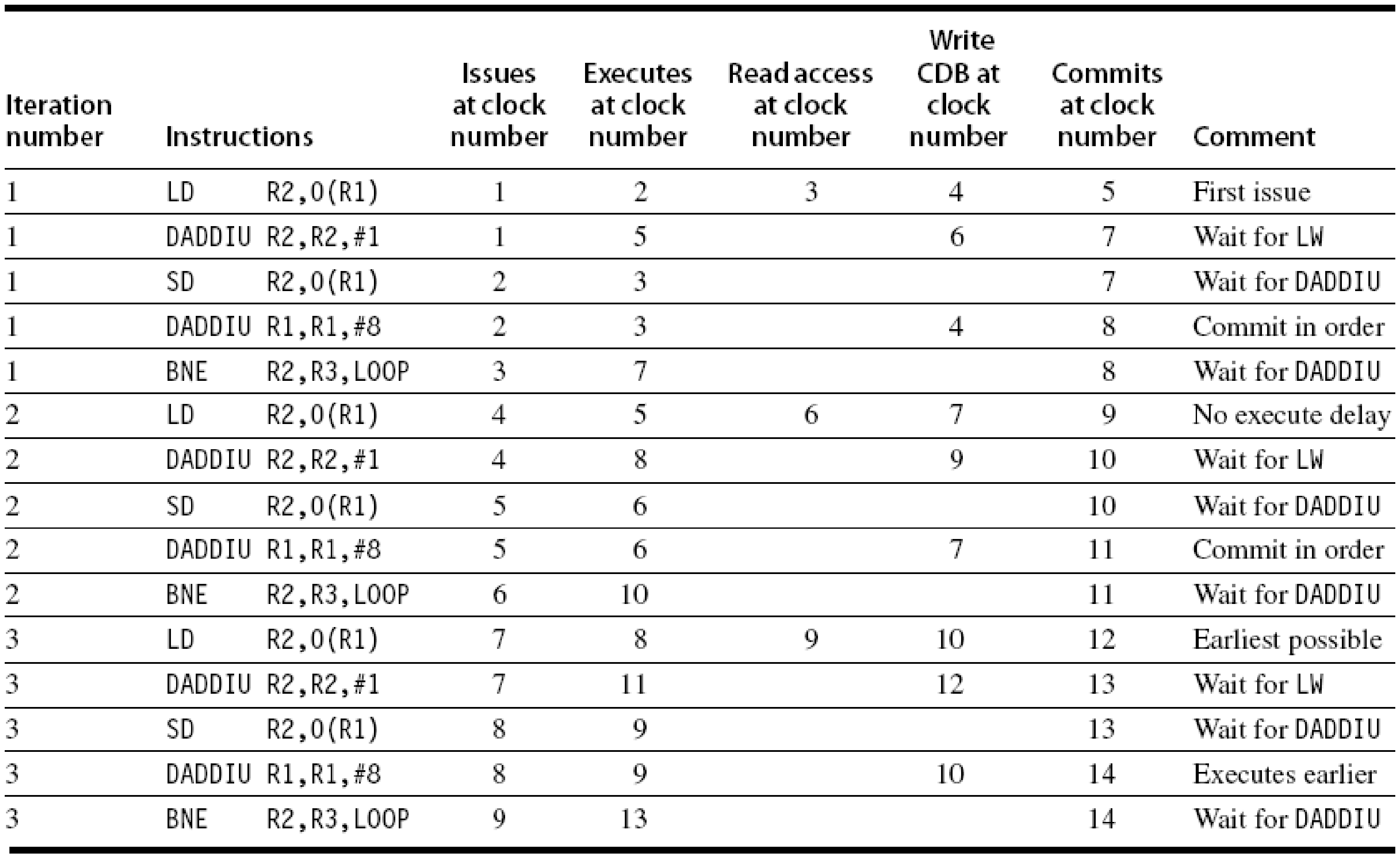
```
Loop:  LD         R2,0(R1)         ;R2=array element
       DADDIU     R2,R2,#1         ;increment R2
       SD         R2,0(R1)         ;store result
       DADDIU     R1,R1,#8         ;increment pointer
       BNE        R2,R3,LOOP       ;branch if not last element
```

Dual issue without 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

- Time of issue, execution, and writing the result for a dual-issue of the pipeline.
- The LD following the BNE (cycles 3, 6) cannot start execution earlier, it must wait until the branch outcome is determined as there is no speculation

Dual issue with speculation



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

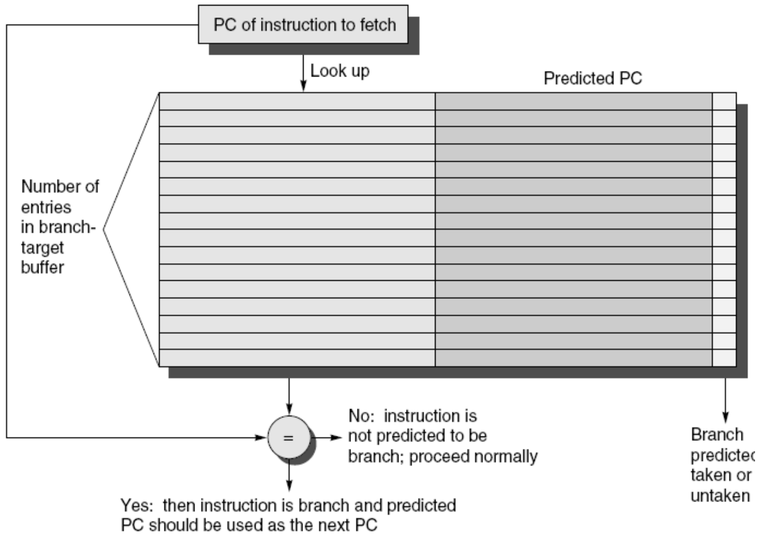
- Time of issue, execution, writing the result, and commit.
- The LD following the BNE can start execution early; speculation is supported.

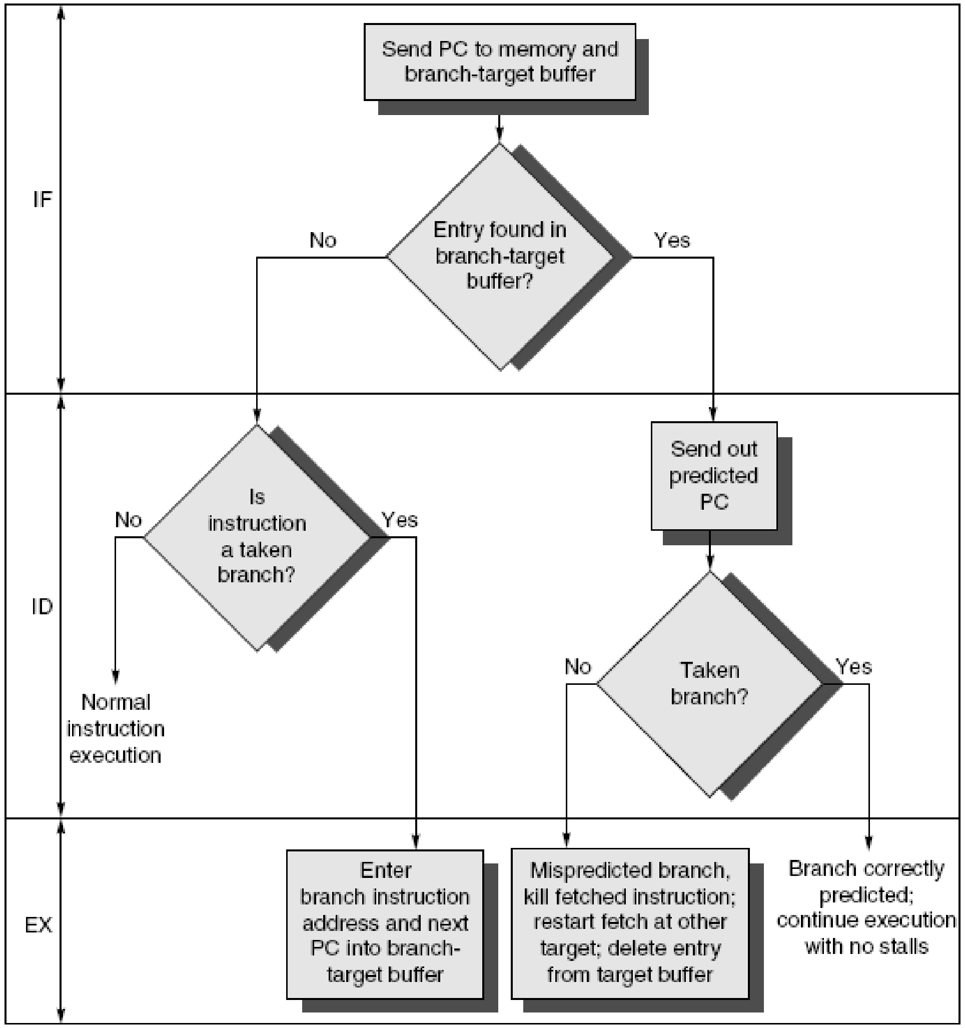
Advanced techniques for instruction delivery

- Increase instruction fetch bandwidth of a multi-issue processor to match the average throughput.
- If an instruction is a branch we will have a zero branch penalty if we know the next PC.
- Requires
  - Wide enough datapaths to IC (Instruction cache)
  - Effective branch handling
- Branch target buffer → cache that stores the predicted address for the next instruction after branch.
- If the PC of a fetched instruction matches the address in this cache → the corresponding predicted PC is used as the next PC.

Branch-target buffer

Next PC prediction buffer, indexed by current PC


PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken


IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

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 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 to speculate?

- 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 used on some processors

Multithreading

- Distinction between TLP (Thread-level parallelism) and multithreading:
  - A multiprocessor → multiple independent threads run concurrently.
  - A uniprocessor system → only one thread can be active at one time
- Multithreading → improve throughput in uniprocessor systems:
  - Shares most of the processor core among the set of threads
  - Duplicates only registers, program counter.
  - Separate page tables for each thread.

Hardware approaches to multithreading

1. Fine-grain multithreading → interleaves execution of multiple threads, switches between threads in each clock cycle, round-robin.
   - Increase in throughput, but slows down execution of each thread.
   - Examples Sun Niagra, Nvidia GPU
2. Coarse grain multithreading → switches only on costly stalls e.g., level 2 or 3 cache misses
   - High pipeline start-up costs
   - No processors
3. Simultaneous multithreading (SMT) → fine-grain multithreading on multiple-issue, dynamically scheduled processor.
   - Most common

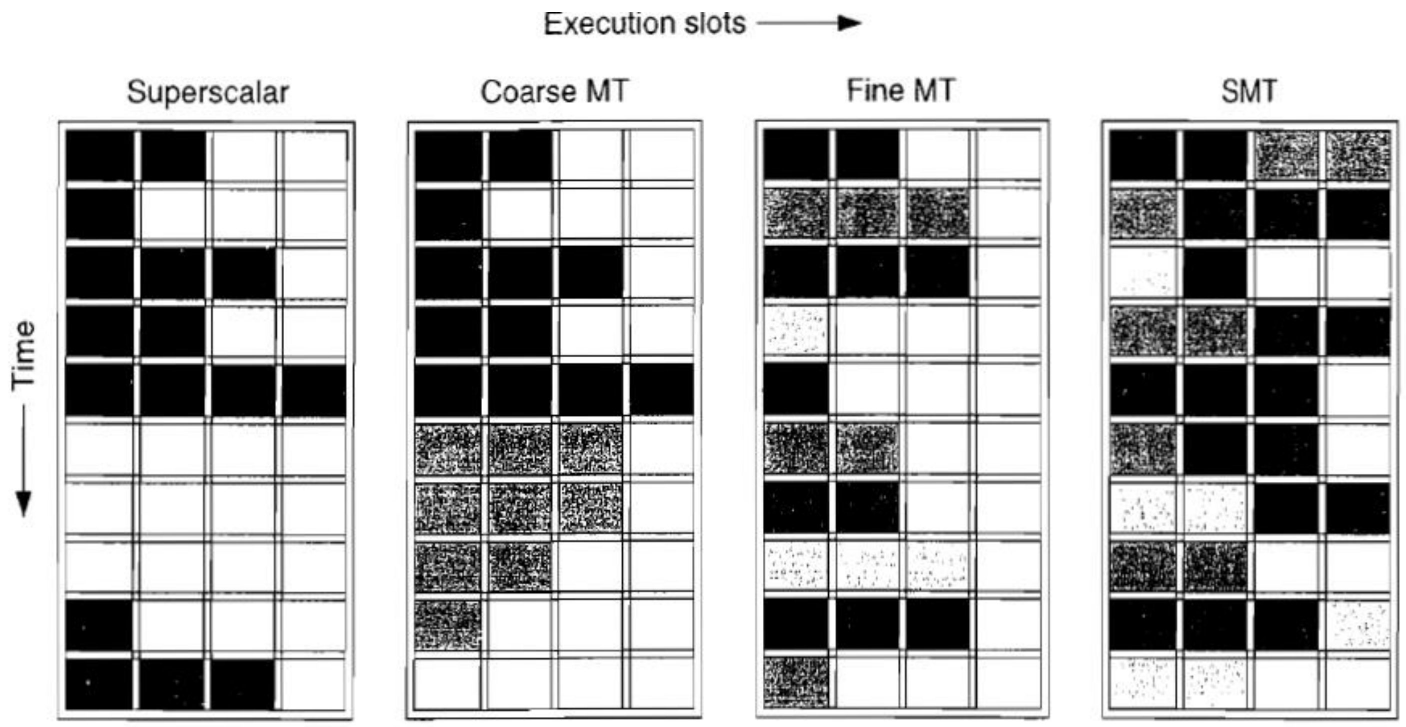


Figure 3.28 How four different approaches use the functional unit execution slots of a superscalar processor. The horizontal dimension represents the instruction execution capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding execution slot is unused in that clock cycle. The shades of gray and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support. The Sun T1 and T2 (aka Niagara) processors are fine-grained multithreaded processors, while the Intel Core i7 and IBM Power7 processors use SMT. The T2 has eight threads, the Power7 has four, and the Intel i7 has two. In all existing SMTs, instructions issue from only one thread at a time. The difference in SMT is that the subsequent decision to execute an instruction is decoupled and could execute the operations coming from several different instructions in the same clock cycle.

ARM Cortex A8

- Dual-issue, statically scheduled superscalar; dynamic issue detection.
- Two instructions per clock cycle, CPI= 0.5.
- Used as basis Apple A9 used in iPad, Intel Core i7 and processors used in smartphones.

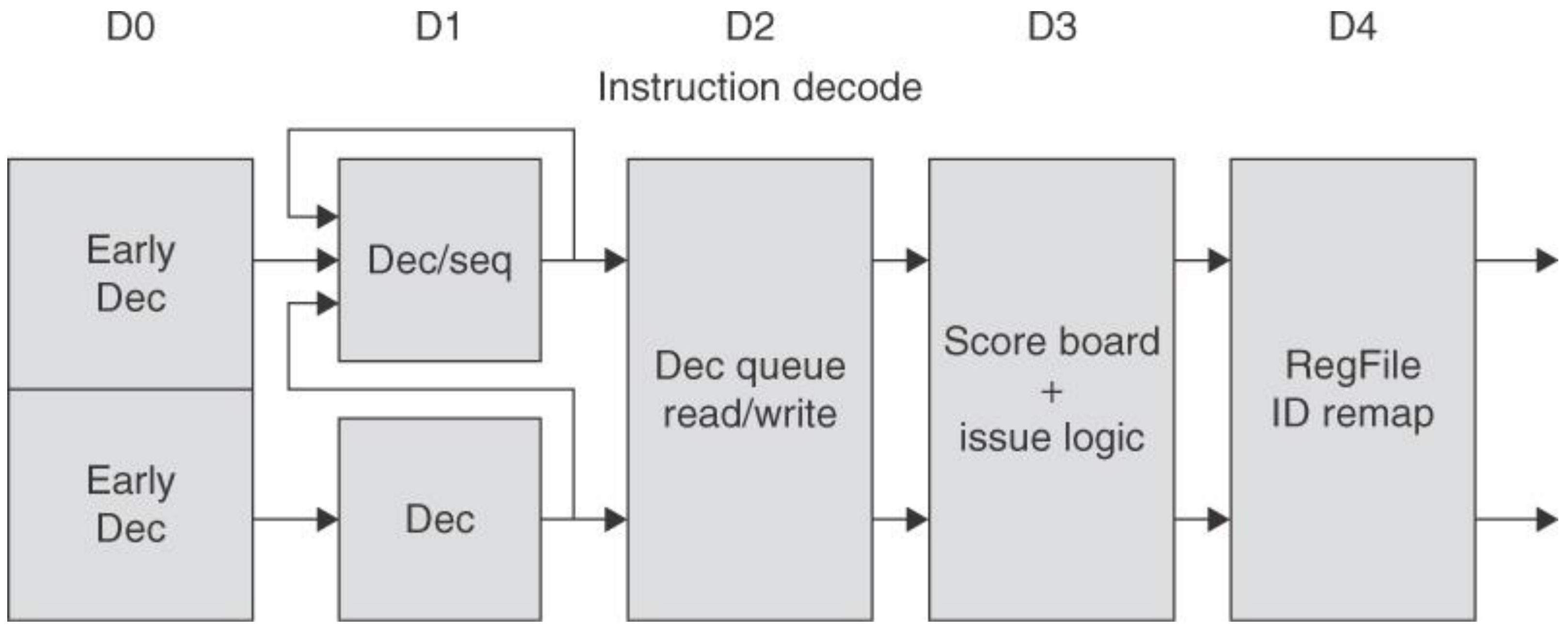


Figure 3.37 The five-stage instruction decode of the A8. In the first stage, a PC produced by the fetch unit (either from the branch target buffer or the PC incrementer) is used to retrieve an 8-byte block from the cache. Up to two instructions are decoded and placed into the decode queue; if neither instruction is a branch, the PC is incremented for the next fetch. Once in the decode queue, the scoreboard logic decides when the instructions can issue. In the issue, the register operands are read; recall that in a simple scoreboard, the operands always come from the registers. The register operands and opcode are sent to the instruction execution portion of the pipeline.

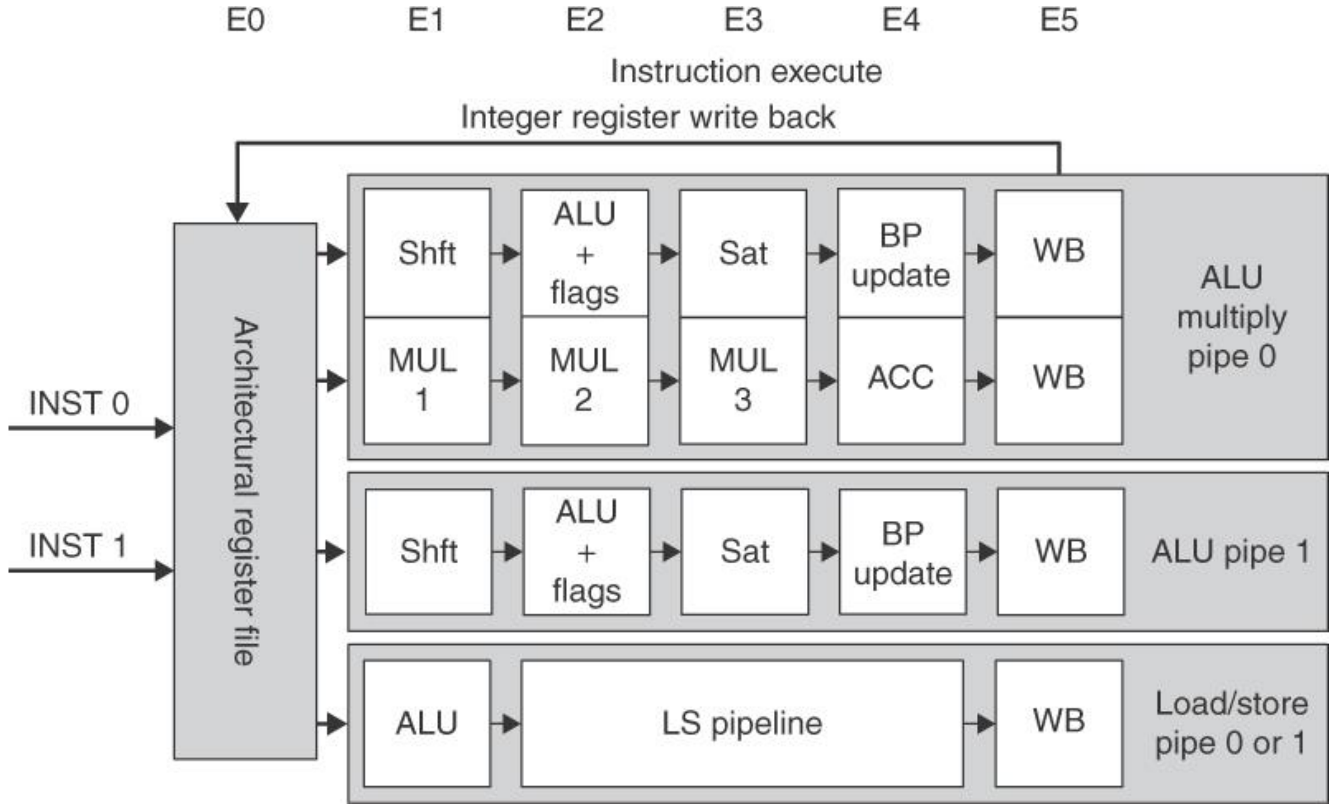


Figure 3.38 The five-stage instruction decode of the A8. Multiply operations are always performed in ALU pipeline 0.

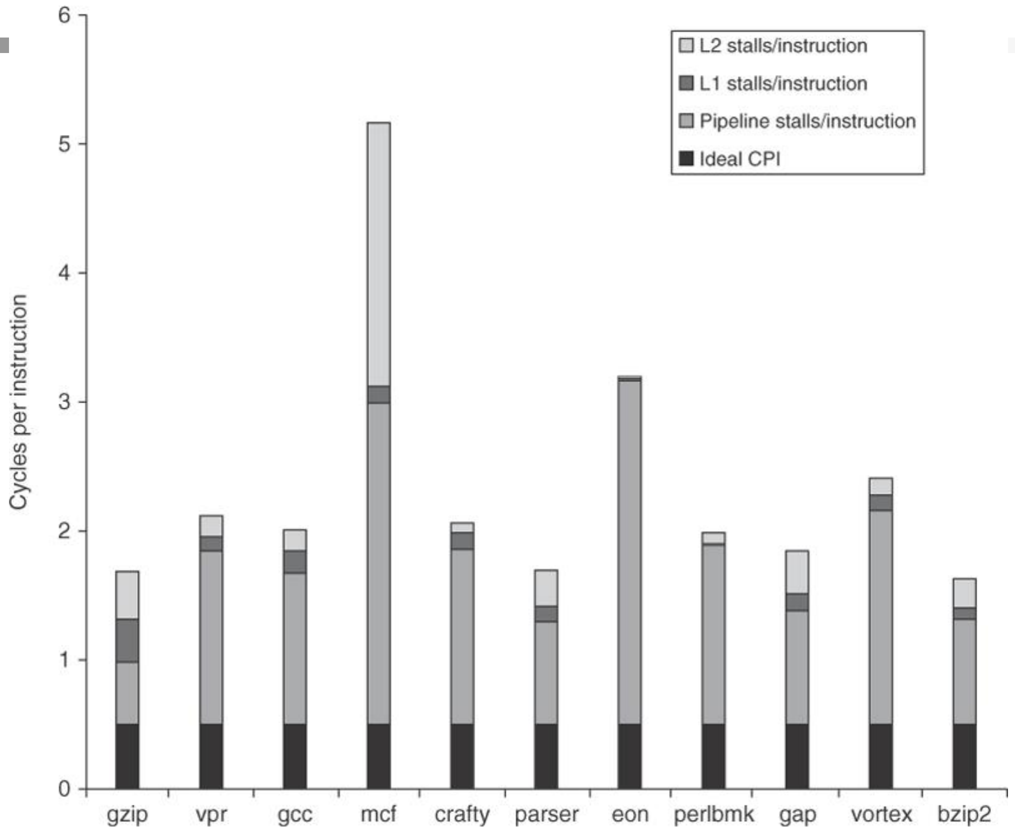


Figure 3.39 The estimated composition of the CPI on the ARM A8 shows that pipeline stalls are the primary addition to the base CPI. eon deserves some special mention, as it does integer-based graphics calculations (ray tracing) and has very few cache misses. It is computationally intensive with heavy use of multiples, and the single multiply pipeline becomes a major bottleneck. This estimate is obtained by using the L1 and L2 miss rates and penalties to compute the L1 and L2 generated stalls per instruction. These are subtracted from the CPI measured by a detailed simulator to obtain the pipeline stalls. Pipeline stalls include all three hazards plus minor effects such as way misprediction.

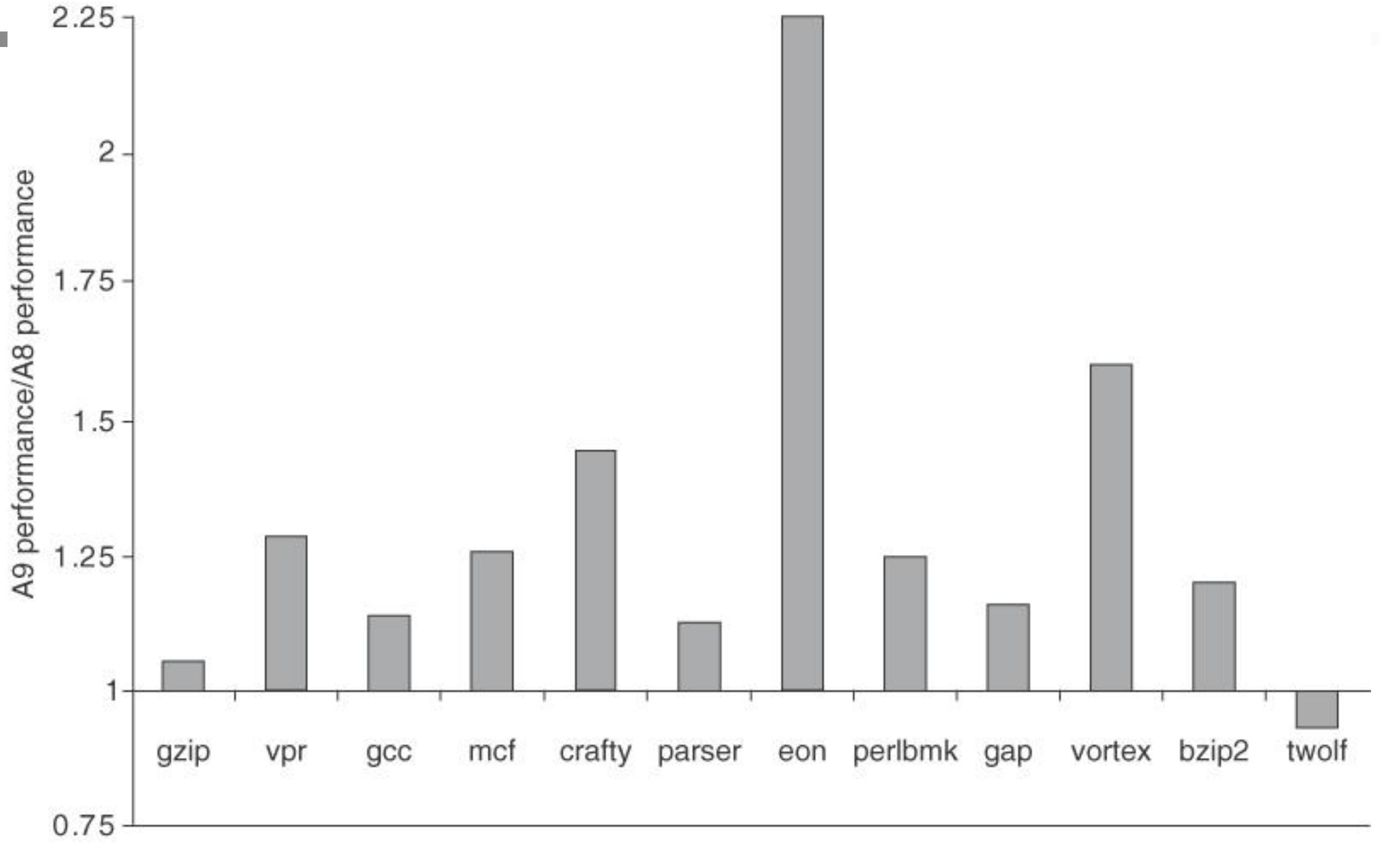


Figure 3.40 The performance ratio for the A9 compared to the A8, both using a 1 GHz clock and the same size caches for L1 and L2, shows that the A9 is about 1.28 times faster. Both runs use a 32 KB primary cache and a 1 MB secondary cache, which is 8-way set associative for the A8 and 16-way for the A9. The block sizes in the caches are 64 bytes for the A8 and 32 bytes for the A9. As mentioned in the caption of Figure 3.39, eon makes intensive use of integer multiply, and the combination of dynamic scheduling and a faster multiply pipeline significantly improves performance on the A9. twolf experiences a small slowdown, likely due to the fact that its cache behavior is worse with the smaller L1 block size of the A9.

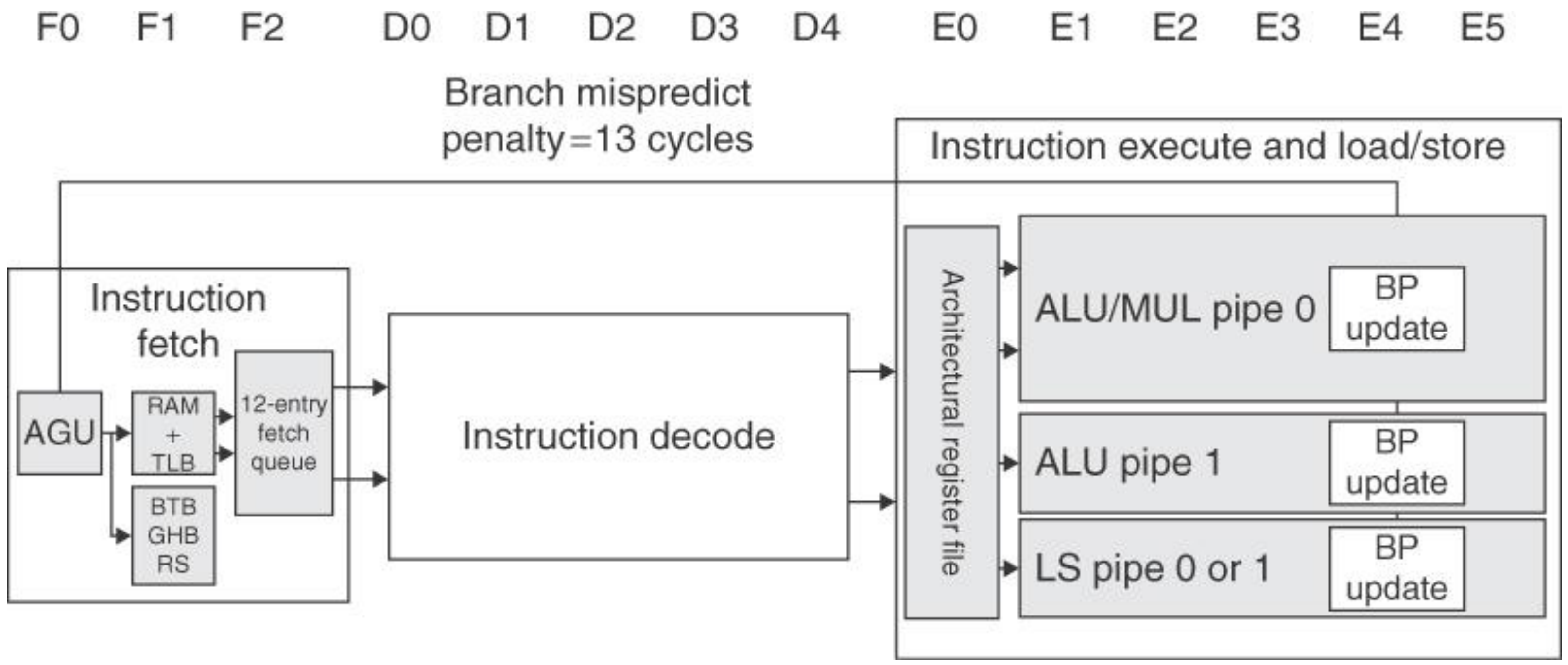


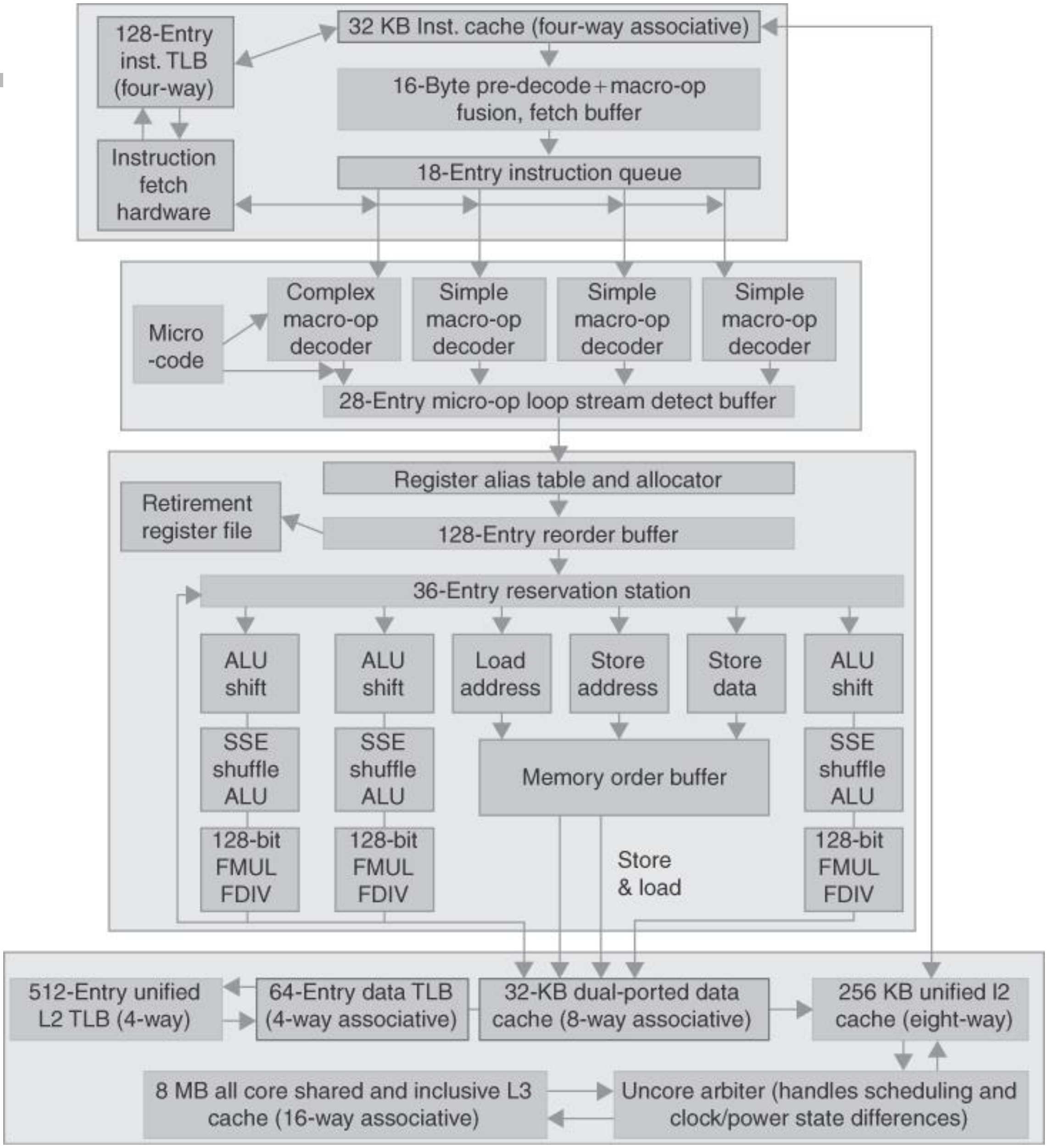
Figure 3.36 The basic structure of the A8 pipeline is 13 stages. Three cycles are used for instruction fetch and four for instruction decode, in addition to a five-cycle integer pipeline. This yields a 13-cycle branch misprediction penalty. The instruction fetch unit tries to keep the 12-entry instruction queue filled.

Intel i7 pipeline

Figure 3.41 The Intel Core i7 pipeline structure shown with the memory system components.

The total pipeline depth is 14 stages, with branch mispredictions costing 17 cycles.

There are 48 load and 32 store buffers. The six independent functional units can each begin execution of a ready micro-op in the same cycle.


128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode+macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
28-Entry micro-op loop stream detect buffer
Register alias table and allocator
Retirement register file
128-Entry reorder buffer
36-Entry reservation station
ALU shift
ALU shift
Load address
Store address
Store data
ALU shift
SSE shuffle ALU
SSE shuffle ALU
Memory order buffer
SSE shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & load
128-bit FMUL FDIV
512-Entry unified L2 TLB (4-way)
64-Entry data TLB (4-way associative)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (eight-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

SPECCPU2006 -benchmark

- CPU2006 is SPEC's next-generation, industry-standardized, CPU-intensive benchmark suite, stressing a system's processor, memory subsystem and compiler.
- Provide a comparative measure of compute-intensive performance across the widest practical range of hardware using workloads developed from real user applications.
- These benchmarks are provided as source code and require the user to be comfortable using compiler commands as well as other commands via a command interpreter using a console or command prompt window in order to generate executable binaries.

SPECCPU2006 - benchmark

- Peralbench →Pearl; cut-down version of Perl v5.8.7
- Mcf → C; vehicle scheduling in public mass transportation. Integer arithmetic.
- Gobmk → C; executes a set of commands to analyze Go positions
- Hmmr → C; uses a Hidden Markov Model for searching gene databases
- Sjeng →C; plays chess and several chess variants
- Libquantum →C; library for the simulation of a quantum computer
- Omnetpp → C++; discrete event simulation of a large Ethernet network
- Astar → C++; derived from a portable 2D path-finding library for game AI
- Xalancbmk→ C++ program; XSLT processor for transforming XML documents into HTML, text, or other XML document types
- Milc → C program for QCD (Quantum Chromodynamics)
- Namd → C++; classical molecular dynamics simulation
- Dealii →C++; PDE solver using the adaptive finite element method
- Soplex →C++; simplex linear program (LP) Solver
- Povray → C++; ray tracing in visualization
- Lbm →C; computational fluid dynamics using the lattice Boltzmann method
- Sphynx3 → C; speech recognition.

Performance of i7

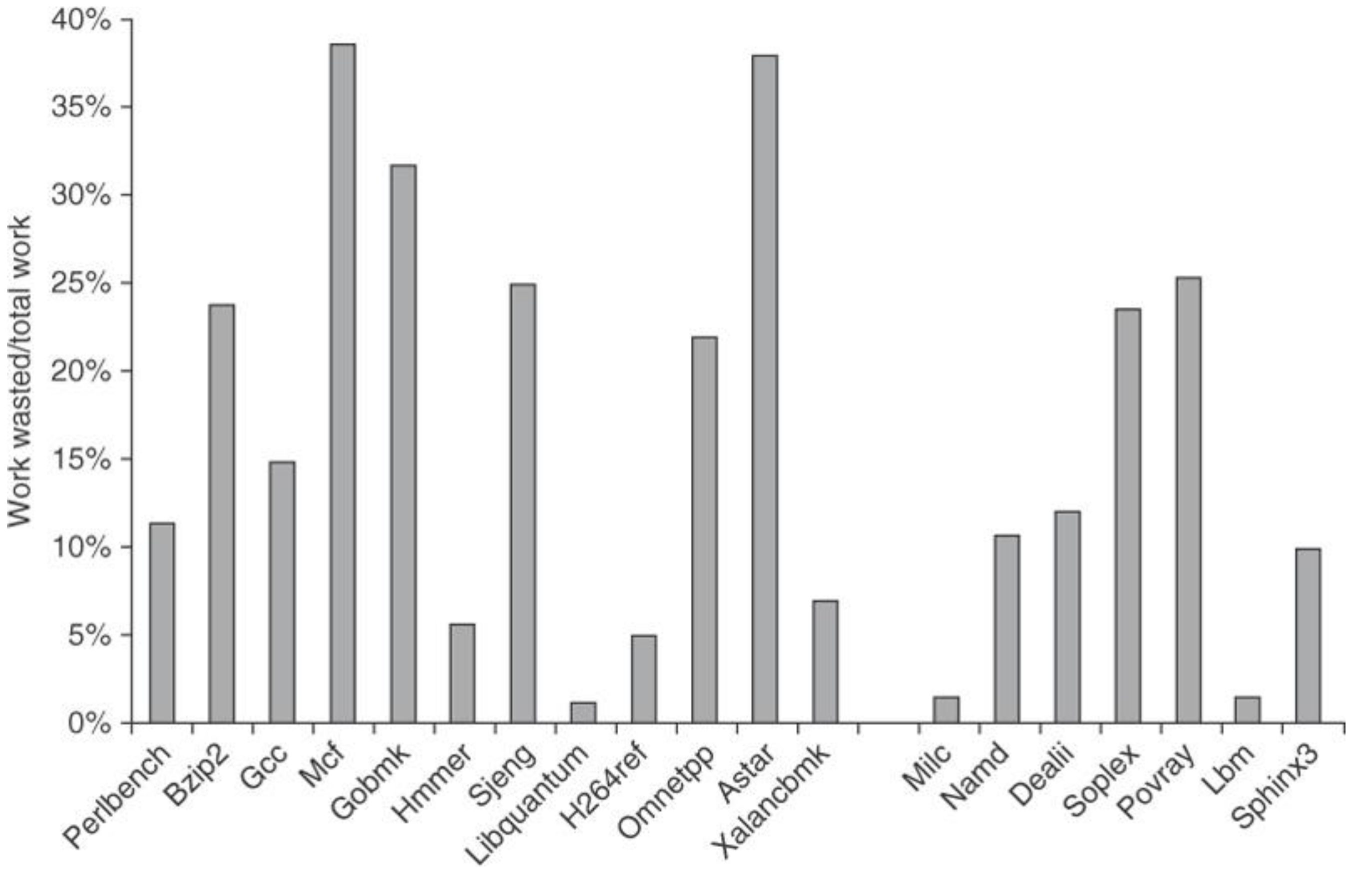


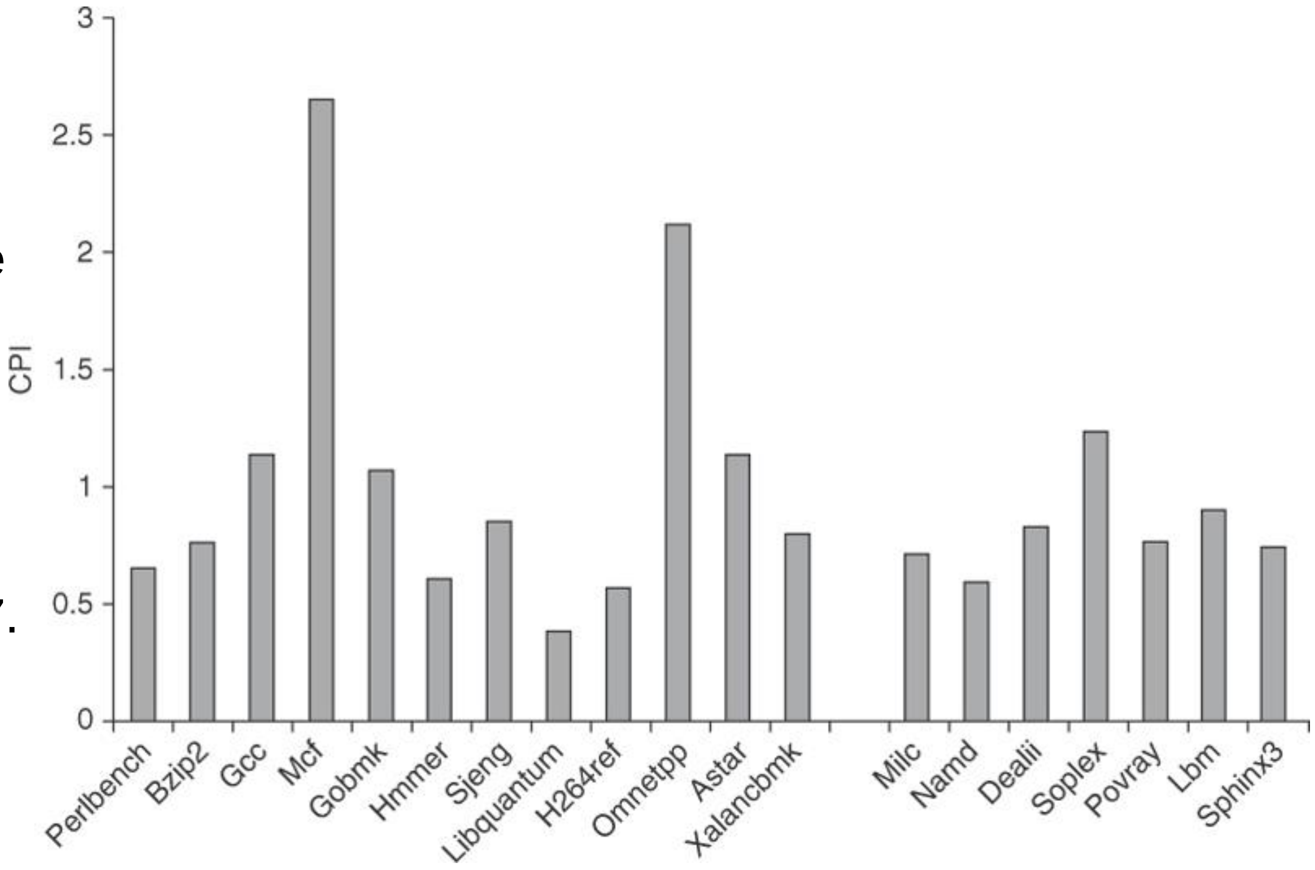
Figure 3.42 The amount of “wasted work” is plotted by taking the ratio of dispatched micro-ops that do not graduate to all dispatched micro-ops. For example, the ratio is 25% for sjeng, meaning that 25% of the dispatched and executed micro-ops are thrown away.

I7 CPI

Figure 3.43 The CPI for the 19 SPECCPU2006 benchmarks shows an average CPI for 0.83 for both the FP and integer benchmarks, although the behaviour is quite different.

A. In the integer case, the CPI values range from 0.44 to 2.66 with a standard deviation of 0.77.

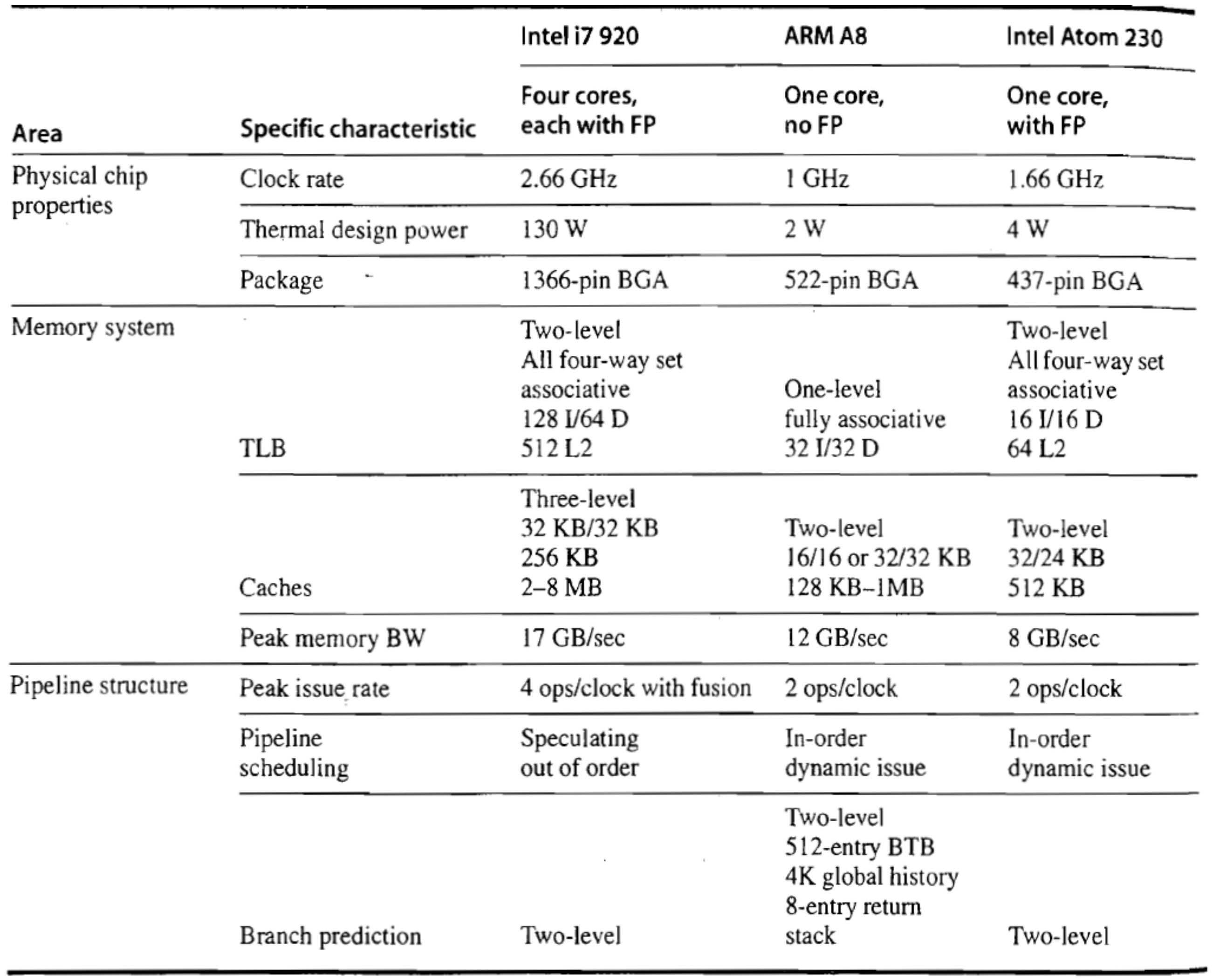
B. The variation in the FP case is from 0.62 to 1.38 with a standard deviation of 0.25.



Fallacies

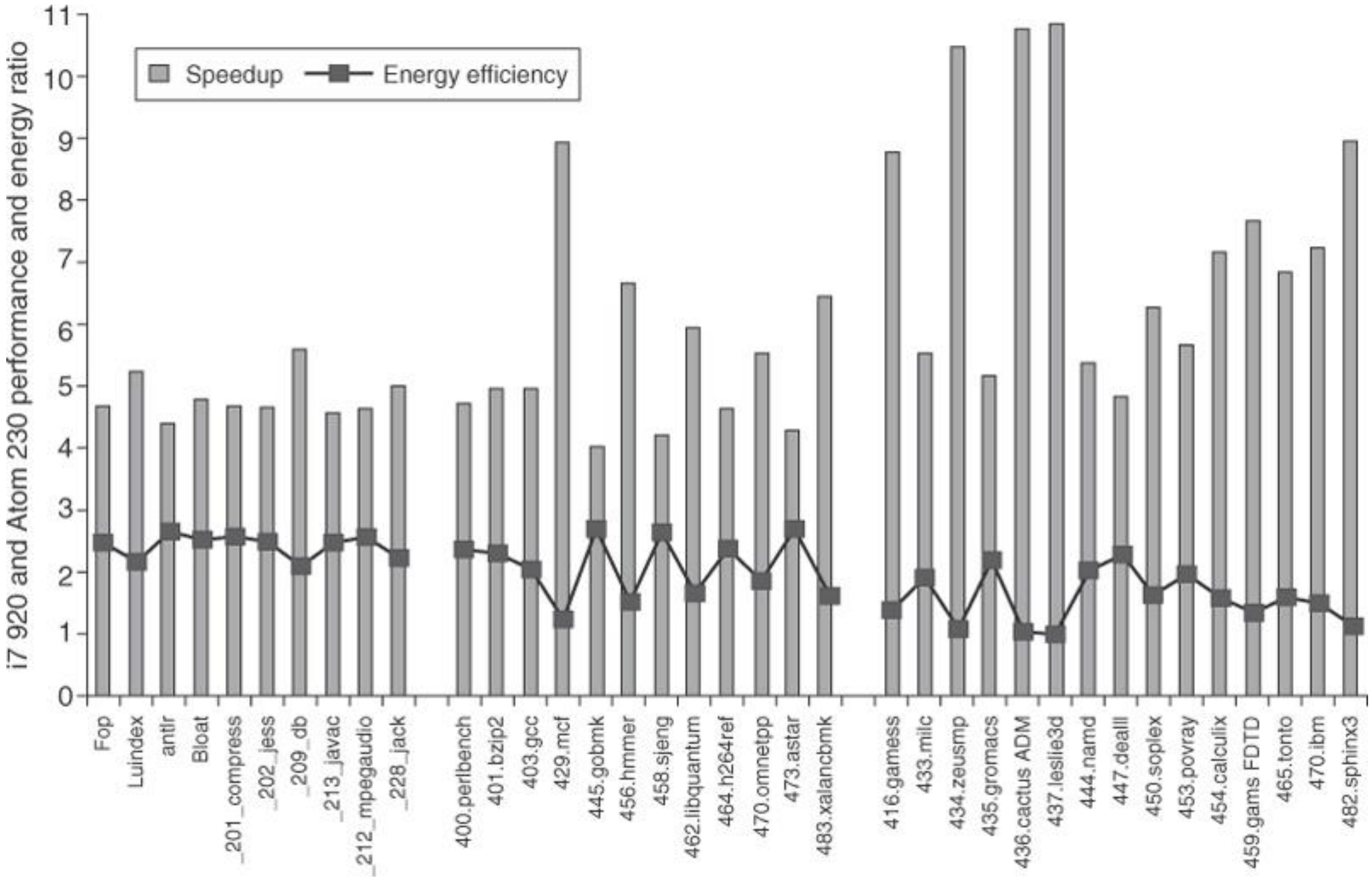
- A. It is easy to predict the performance and the energy efficiency of two different versions of the same instruction set architecture, if we hold the technology constant. Case in point Intel i7 920, Intel Atom 230, and ARM A8

Figure 3.45 The relative performance and energy efficiency for a set of single-threaded benchmarks shows the i7 920 is 4 to over 10 times faster than the Atom 230 but that it is about 2 times *less* power efficient on average! Performance is shown in the columns as i7 relative to Atom, which is execution time (i7)/execution time (Atom). Energy is shown with the line as Energy (Atom)/Energy (i7). The i7 never beats the Atom in energy efficiency, although it is essentially as good on four benchmarks, three of which are floating point. Only one core is active on the i7, and the rest are in deep power saving mode. Turbo Boost is used on the i7, which increases its performance advantage but slightly decreases its relative energy efficiency.



		Intel i7 920	ARM A8	Intel Atom 230
Area	**Specific characteristic**	**Four cores, each with FP**	**One core, no FP**	**One core, with FP**
Physical chip properties	Clock rate	2.66 GHz	1 GHz	1.66 GHz
	Thermal design power	130 W	2 W	4 W
	Package	1366-pin BGA	522-pin BGA	437-pin BGA
Memory system	TLB	Two-level All four-way set associative 128 I/64 D 512 L2	One-level fully associative 32 I/32 D	Two-level All four-way set associative 16 I/16 D 64 L2
	Caches	Three-level 32 KB/32 KB 256 KB 2–8 MB	Two-level 16/16 or 32/32 KB 128 KB–1MB	Two-level 32/24 KB 512 KB
	Peak memory BW	17 GB/sec	12 GB/sec	8 GB/sec
Pipeline structure	Peak issue rate	4 ops/clock with fusion	2 ops/clock	2 ops/clock
	Pipeline scheduling	Speculating out of order	In-order dynamic issue	In-order dynamic issue
	Branch prediction	Two-level	Two-level 512-entry BTB 4K global history 8-entry return stack	Two-level

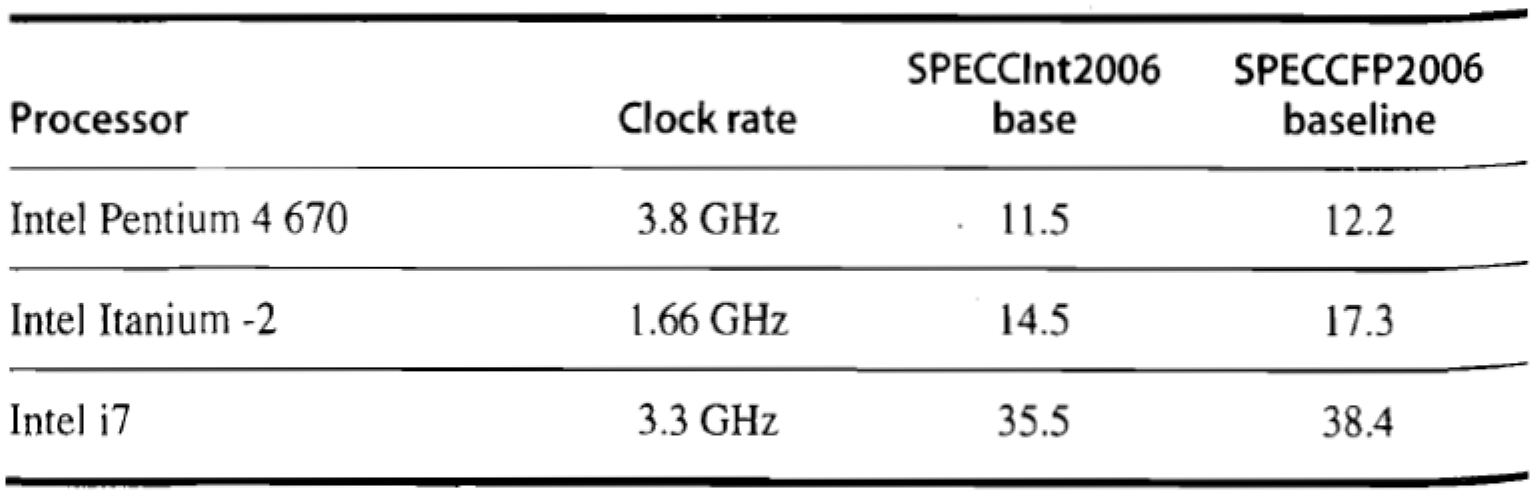
Figure 3.44 An overview of the four-core Intel i7 920, an example of a typical Arm A8 processor chip (with a 256 MB L2, 32K L1s, and no floating point), and the Intel ARM 230 clearly showing the difference in design philosophy between a processor intended for the PMD (in the case of ARM) or netbook space (in the case of Atom) and a processor for use in servers and high-end desktops. Remember, the i7 includes four cores, each of which is several times higher in performance than the one-core A8 or Atom. All these processors are implemented in a comparable 45 nm technology.



i7 920 and Atom 230 performance and energy ratio
Speedup
Energy efficiency
0
1
2
3
4
5
6
7
8
9
10
11
Fop
Luindex
antlr
Bloat
_201_compress
_202_jess
_209_db
_213_javac
_212_mpegaudio
_228_jack
400.perlbench
401.bzip2
403.gcc
429.mcf
445.gobmk
456.hmmer
458.sjeng
462.libquantum
464.h264ref
470.omnetpp
473.astar
483.xalancbmk
416.gamess
433.milc
434.zeusmp
435.gromacs
436.cactus ADM
437.leslie3d
444.namd
447.dealII
450.soplex
453.povray
454.calculix
459.gams FDTD
465.tonto
470.ibm
482.sphinx3

Fallacies

- B. Processors with lower CPI will always be faster.
- C. Processors with faster clock rate will always be faster.



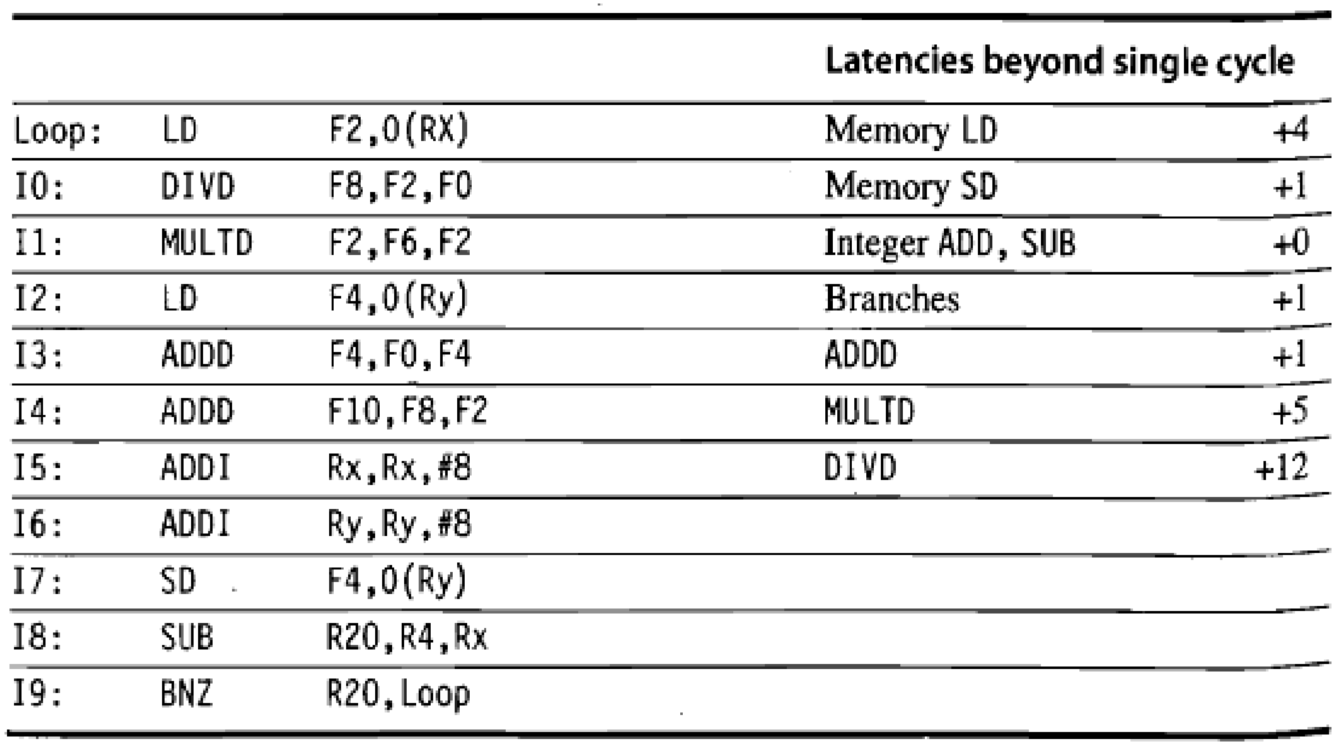
Processor	Clock rate	SPECCInt2006 base	SPECCFP2006 baseline
Intel Pentium 4 670	3.8 GHz	11.5	12.2
Intel Itanium -2	1.66 GHz	14.5	17.3
Intel i7	3.3 GHz	35.5	38.4

Figure 3.46 Three different Intel processors vary widely. Although the Itanium processor has two cores and the i7 four, only one core is used in the benchmarks.

The product of the CPI and the clock rate determine performance!!

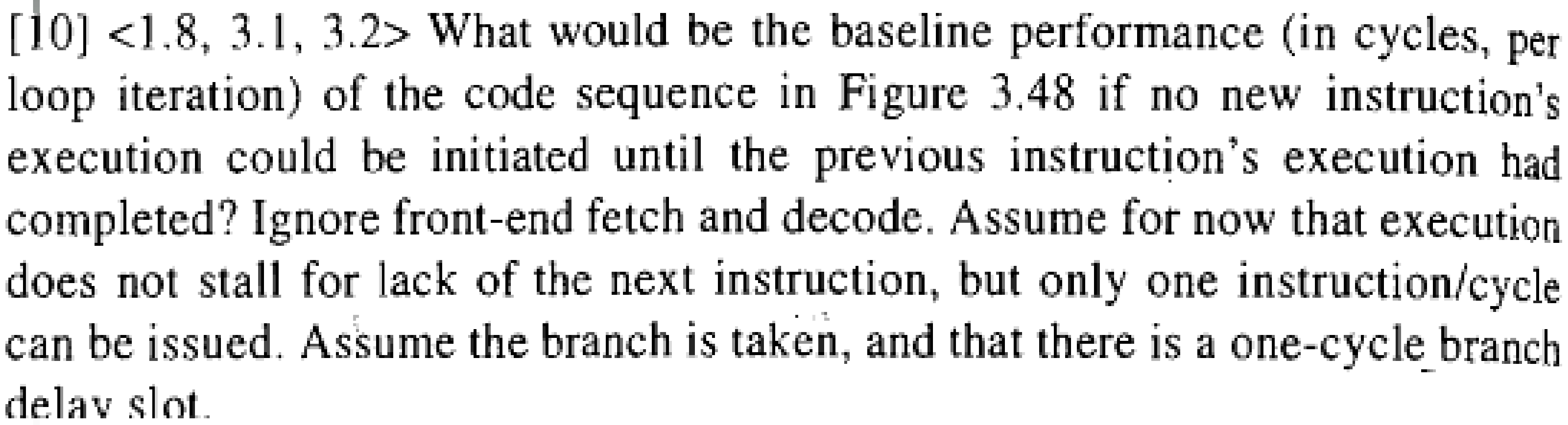
Problem solving

- Consider the following code; the latency of all the instructions used by the code snippet are also shown in Figure 3.48 below



			Latencies beyond single cycle	
Loop:	LD	F2,0(RX)	Memory LD	+4
I0:	DIVD	F8,F2,F0	Memory SD	+1
I1:	MULTD	F2,F6,F2	Integer ADD, SUB	+0
I2:	LD	F4,0(Ry)	Branches	+1
I3:	ADDD	F4,F0,F4	ADDD	+1
I4:	ADDD	F10,F8,F2	MULTD	+5
I5:	ADDI	Rx,Rx,#8	DIVD	+12
I6:	ADDI	Ry,Ry,#8		
I7:	SD	F4,0(Ry)		
I8:	SUB	R20,R4,Rx		
I9:	BNZ	R20,Loop		

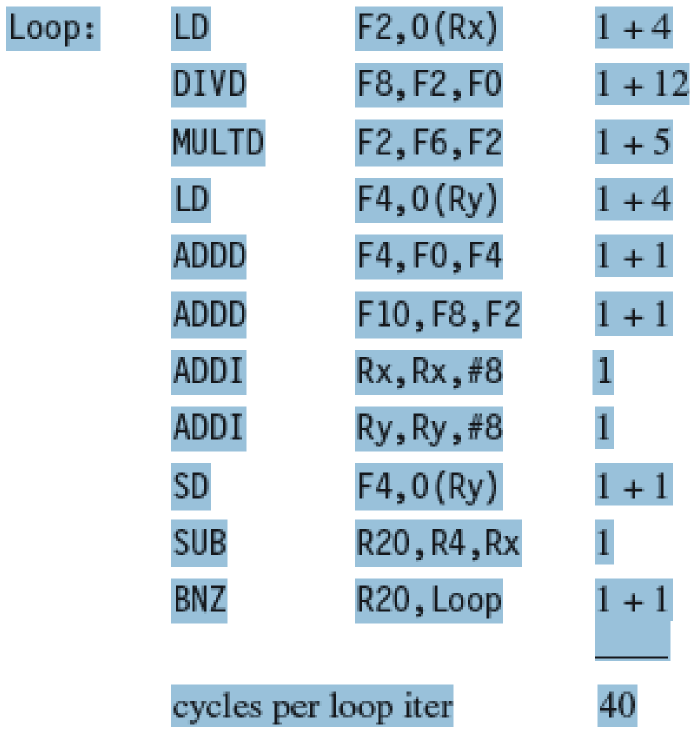
Problem 1



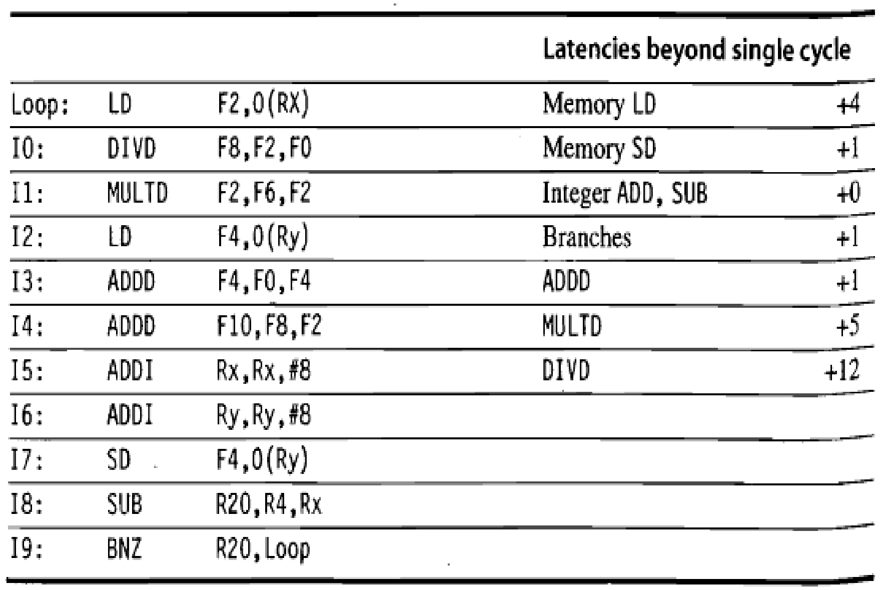
[10] <1.8, 3.1, 3.2> What would be the baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.48 if no new instruction's execution could be initiated until the previous instruction's execution had completed? Ignore front-end fetch and decode. Assume for now that execution does not stall for lack of the next instruction, but only one instruction/cycle can be issued. Assume the branch is taken, and that there is a one-cycle branch delay slot.

- When a branch instruction is involved, the location of the following delay slot instruction in the pipeline may be called a **branch delay slot**. Branch delay slots are found mainly in DSP architectures and older RISC architectures. MIPS, PA-RISC, ETRAX CRIS, SuperH, and SPARC are RISC architectures that each have a single branch delay slot.

P1 - solution



Loop:	LD	F2,0(Rx)	1 + 4
	DIVD	F8,F2,F0	1 + 12
	MULTD	F2,F6,F2	1 + 5
	LD	F4,0(Ry)	1 + 4
	ADDD	F4,F0,F4	1 + 1
	ADDD	F10,F8,F2	1 + 1
	ADDI	Rx,Rx,#8	1
	ADDI	Ry,Ry,#8	1
	SD	F4,0(Ry)	1 + 1
	SUB	R20,R4,Rx	1
	BNZ	R20,Loop	1 + 1
	cycles per loop iter		40



			Latencies beyond single cycle	
Loop:	LD	F2,0(RX)	Memory LD	+4
I0:	DIVD	F8,F2,F0	Memory SD	+1
I1:	MULTD	F2,F6,F2	Integer ADD, SUB	+0
I2:	LD	F4,0(Ry)	Branches	+1
I3:	ADDD	F4,F0,F4	ADDD	+1
I4:	ADDD	F10,F8,F2	MULTD	+5
I5:	ADDI	Rx,Rx,#8	DIVD	+12
I6:	ADDI	Ry,Ry,#8		
I7:	SD	F4,0(Ry)		
I8:	SUB	R20,R4,Rx		
I9:	BNZ	R20,Loop		

Each instruction requires one clock cycle of execution (a clock cycle in which that instruction, and only that instruction, is occupying the execution units; since every instruction must execute, the loop will take at least that many clock cycles). To that base number, we add the extra latency cycles. Don't forget the branch shadow cycle.

Problem 2

[10] <1.8, 3.1, 3.2> Think about what latency numbers really mean—they indicate the number of cycles a given function requires to produce its output, nothing more. If the overall pipeline stalls for the latency cycles of each functional unit, then you are at least guaranteed that any pair of back-to-back instructions (a "producer" followed by a "consumer") will execute correctly. But not all instruction pairs have a producer/consumer relationship. Sometimes two adjacent instructions have nothing to do with each other. How many cycles would the loop body in the code sequence in Figure 3.48 require if the pipeline detected true data dependences and only stalled on those, rather than blindly stalling everything just because one functional unit is busy? Show the code with `<stall>` inserted where necessary to accommodate stated latencies. (*Hint:* An instruction with latency +2 requires two `<stall>` cycles to be inserted into the code sequence. Think of it this way: A one-cycle instruction has latency 1 + 0, meaning zero extra wait states. So, latency 1 + 1 implies one stall cycle; latency 1 + *N* has *N* extra stall cycles.

P2 - solution

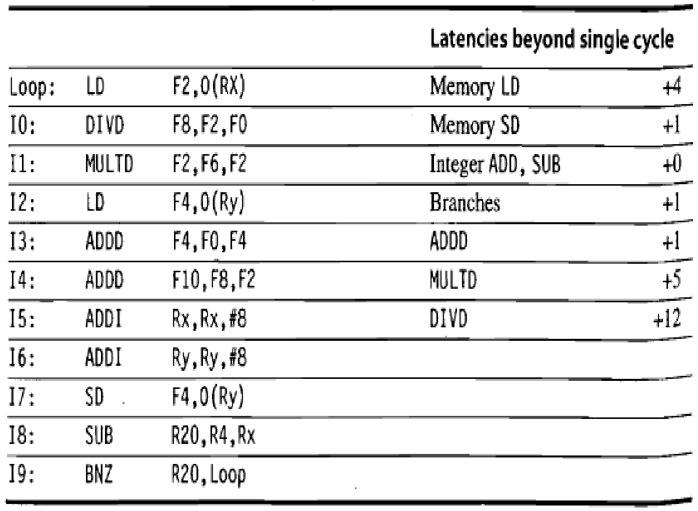
			Latencies beyond single cycle	
Loop:	LD	F2,0(RX)	Memory LD	+4
I0:	DIVD	F8,F2,F0	Memory SD	+1
I1:	MULTD	F2,F6,F2	Integer ADD, SUB	+0
I2:	LD	F4,0(Ry)	Branches	+1
I3:	ADDD	F4,F0,F4	ADDD	+1
I4:	ADDD	F10,F8,F2	MULTD	+5
I5:	ADDI	Rx,Rx,#8	DIVD	+12
I6:	ADDI	Ry,Ry,#8		
I7:	SD	F4,0(Ry)		
I8:	SUB	R20,R4,Rx		
I9:	BNZ	R20,Loop		

```
Loop:   LD          F2,0(Rx)        1 + 4
        <stall>
        <stall>
        <stall>
        <stall>
        DIVD        F8,F2,F0        1 + 12
        MULTD       F2,F6,F2        1 + 5
        LD          F4,0(Ry)        1 + 4
        <stall due to LD latency>
        <stall due to LD latency>
        <stall due to LD latency>
        <stall due to LD latency>
        ADDD        F4,F0,F4        1 + 1
        <stall due to ADDD latency>
        <stall due to DIVD latency>
        <stall due to DIVD latency>
        <stall due to DIVD latency>
        <stall due to DIVD latency>
        ADDD        F10,F8,F2       1 + 1
        ADDI        Rx,Rx,#8        1
        ADDI        Ry,Ry,#8        1
        SD          F4,0(Ry)        1 + 1
        SUB         R20,R4,Rx       1
        BNZ         R20,Loop        1 + 1
        <stall branch delay slot>
                                    ------
        cycles per loop iter        25
```

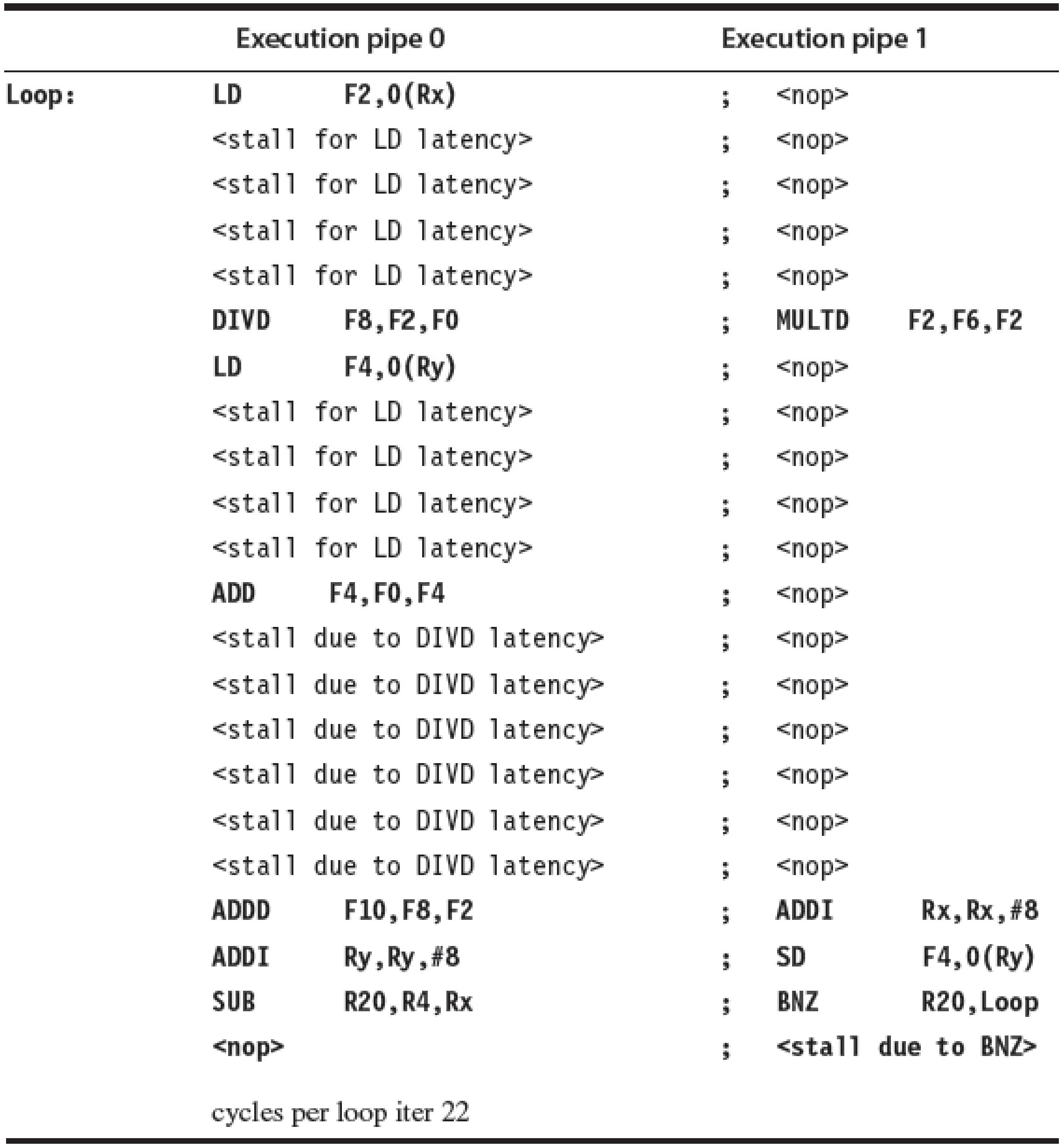
Problem 3

[15] <3.6, 3.7> Consider a multiple-issue design. Suppose you have two execution pipelines, each capable of beginning execution of one instruction per cycle, and enough fetch/decode bandwidth in the front end so that it will not stall your execution. Assume results can be immediately forwarded from one execution unit to another, or to itself. Further assume that the only reason an execution pipeline would stall is to observe a true data dependency. Now how many cycles does the loop require?

P3 -solution

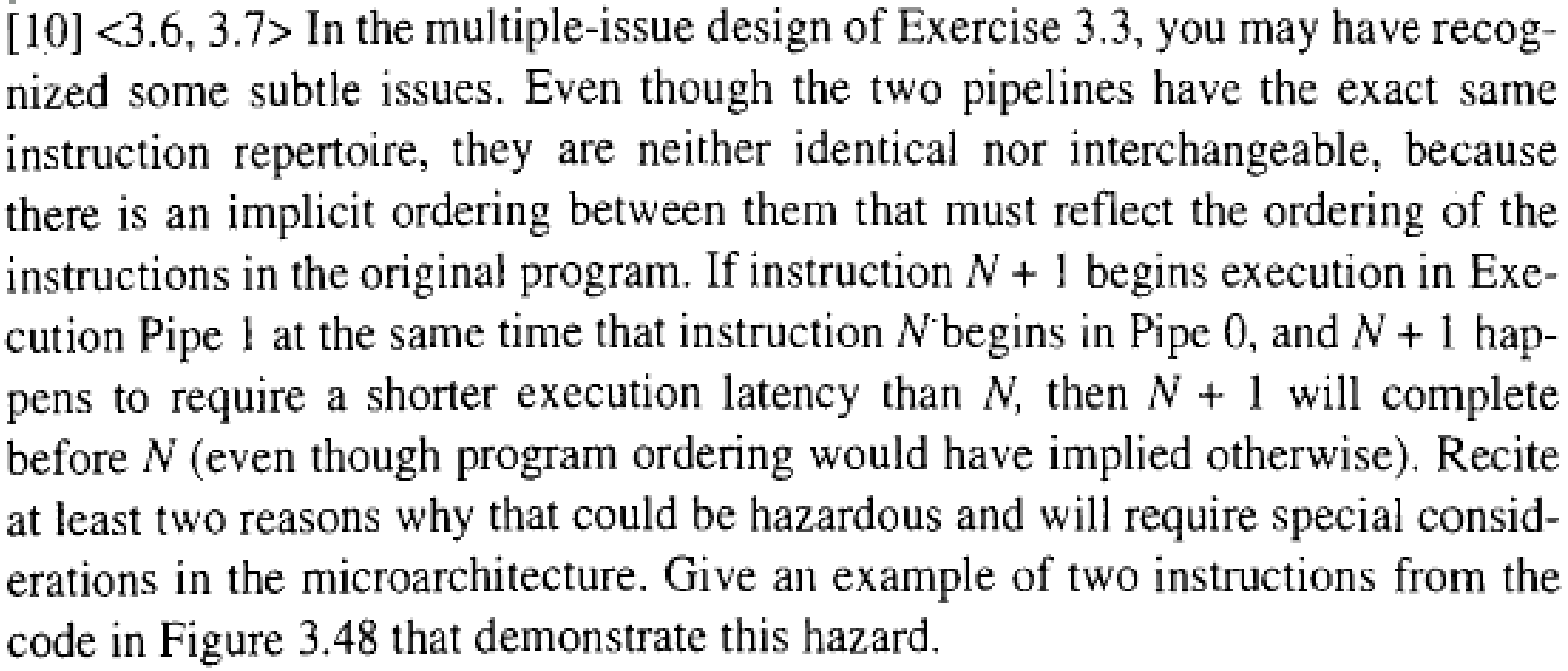


			Latencies beyond single cycle	
Loop:	LD	F2,0(RX)	Memory LD	+4
I0:	DIVD	F8,F2,F0	Memory SD	+1
I1:	MULTD	F2,F6,F2	Integer ADD, SUB	+0
I2:	LD	F4,0(Ry)	Branches	+1
I3:	ADDD	F4,F0,F4	ADDD	+1
I4:	ADDD	F10,F8,F2	MULTD	+5
I5:	ADDI	Rx,Rx,#8	DIVD	+12
I6:	ADDI	Ry,Ry,#8		
I7:	SD	F4,0(Ry)		
I8:	SUB	R20,R4,Rx		
I9:	BNZ	R20,Loop		



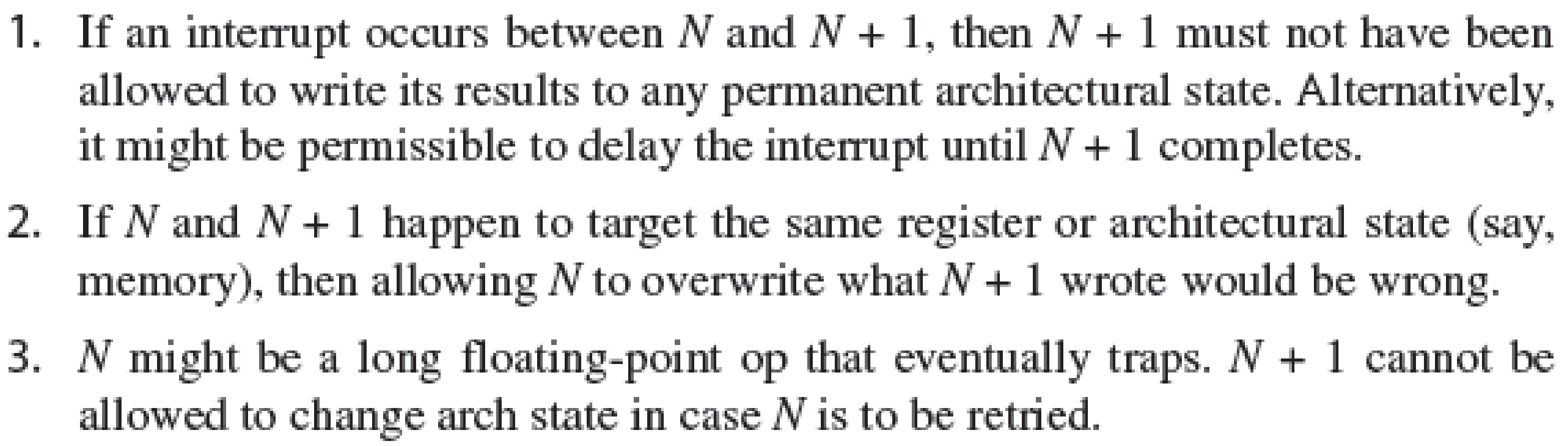
	Execution pipe 0			Execution pipe 1	
Loop:	LD	F2,0(Rx)	;	<nop>	
	<stall for LD latency>		;	<nop>	
	<stall for LD latency>		;	<nop>	
	<stall for LD latency>		;	<nop>	
	<stall for LD latency>		;	<nop>	
	DIVD	F8,F2,F0	;	MULTD	F2,F6,F2
	LD	F4,0(Ry)	;	<nop>	
	<stall for LD latency>		;	<nop>	
	<stall for LD latency>		;	<nop>	
	<stall for LD latency>		;	<nop>	
	<stall for LD latency>		;	<nop>	
	ADD	F4,F0,F4	;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	ADDD	F10,F8,F2	;	ADDI	Rx,Rx,#8
	ADDI	Ry,Ry,#8	;	SD	F4,0(Ry)
	SUB	R20,R4,Rx	;	BNZ	R20,Loop
	<nop>		;	<stall due to BNZ>	
	cycles per loop iter 22				

Problem 4



[10] <3.6, 3.7> In the multiple-issue design of Exercise 3.3, you may have recognized some subtle issues. Even though the two pipelines have the exact same instruction repertoire, they are neither identical nor interchangeable, because there is an implicit ordering between them that must reflect the ordering of the instructions in the original program. If instruction *N* + 1 begins execution in Execution Pipe 1 at the same time that instruction *N* begins in Pipe 0, and *N* + 1 happens to require a shorter execution latency than *N*, then *N* + 1 will complete before *N* (even though program ordering would have implied otherwise). Recite at least two reasons why that could be hazardous and will require special considerations in the microarchitecture. Give an example of two instructions from the code in Figure 3.48 that demonstrate this hazard.

P4 - solution



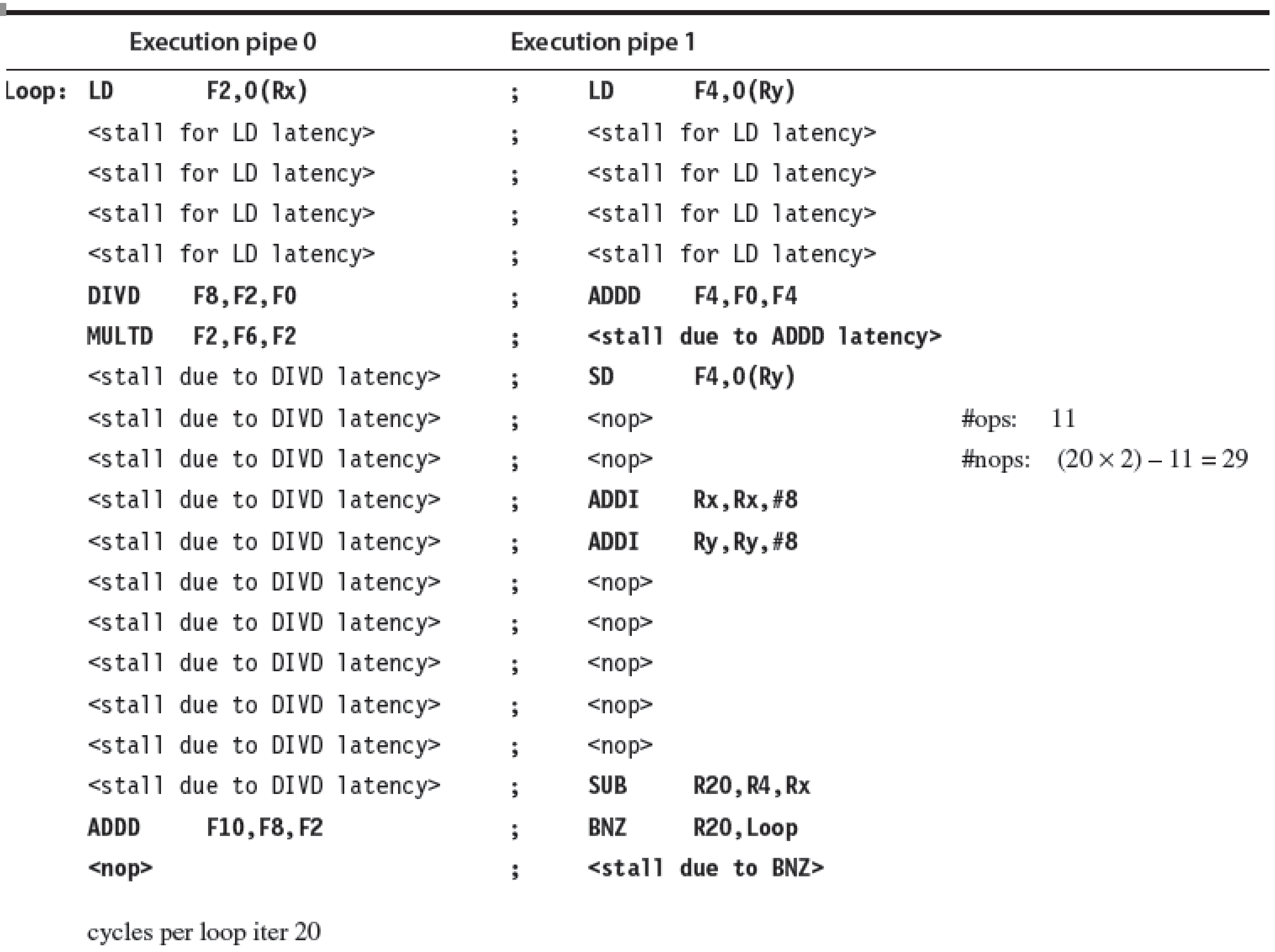
1. If an interrupt occurs between N and $N + 1$, then $N + 1$ must not have been allowed to write its results to any permanent architectural state. Alternatively, it might be permissible to delay the interrupt until $N + 1$ completes.
2. If N and $N + 1$ happen to target the same register or architectural state (say, memory), then allowing N to overwrite what $N + 1$ wrote would be wrong.
3. N might be a long floating-point op that eventually traps. $N + 1$ cannot be allowed to change arch state in case N is to be retried.

Long-latency ops are at highest risk of being passed by a subsequent op. The DIVD instr will complete long after the LD F4,0(Ry), for example.

Problem 5

[20] <3.7> Reorder the instructions to improve performance of the code in Figure 3.48. Assume the two-pipe machine in Exercise 3.3 and that the out-of-order completion issues of Exercise 3.4 have been dealt with successfully. Just worry about observing true data dependences and functional unit latencies for now. How many cycles does your reordered code take?

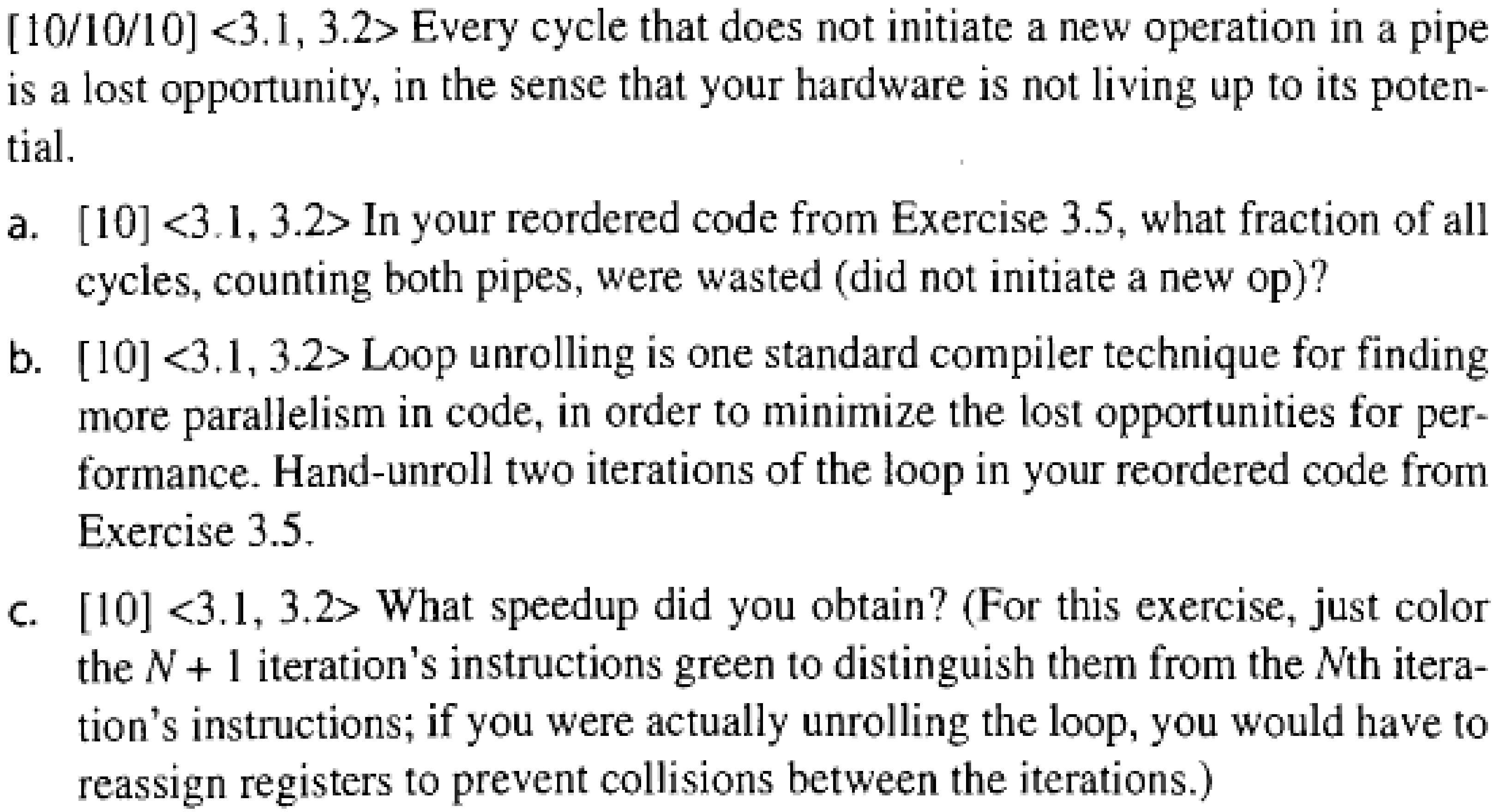
P5-solution



```
                Execution pipe 0                      Execution pipe 1
Loop:  LD       F2,0(Rx)                       ;     LD       F4,0(Ry)
       <stall for LD latency>                  ;     <stall for LD latency>
       <stall for LD latency>                  ;     <stall for LD latency>
       <stall for LD latency>                  ;     <stall for LD latency>
       <stall for LD latency>                  ;     <stall for LD latency>
       DIVD     F8,F2,F0                       ;     ADDD     F4,F0,F4
       MULTD    F2,F6,F2                       ;     <stall due to ADDD latency>
       <stall due to DIVD latency>             ;     SD       F4,0(Ry)
       <stall due to DIVD latency>             ;     <nop>                    #ops:   11
       <stall due to DIVD latency>             ;     <nop>                    #nops:  (20 × 2) – 11 = 29
       <stall due to DIVD latency>             ;     ADDI     Rx,Rx,#8
       <stall due to DIVD latency>             ;     ADDI     Ry,Ry,#8
       <stall due to DIVD latency>             ;     <nop>
       <stall due to DIVD latency>             ;     <nop>
       <stall due to DIVD latency>             ;     <nop>
       <stall due to DIVD latency>             ;     <nop>
       <stall due to DIVD latency>             ;     <nop>
       <stall due to DIVD latency>             ;     SUB      R20,R4,Rx
       ADDD     F10,F8,F2                      ;     BNZ      R20,Loop
       <nop>                                   ;     <stall due to BNZ>

       cycles per loop iter 20
```

Problem 6



[10/10/10] <3.1, 3.2> Every cycle that does not initiate a new operation in a pipe is a lost opportunity, in the sense that your hardware is not living up to its potential.

a. [10] <3.1, 3.2> In your reordered code from Exercise 3.5, what fraction of all cycles, counting both pipes, were wasted (did not initiate a new op)?

b. [10] <3.1, 3.2> Loop unrolling is one standard compiler technique for finding more parallelism in code, in order to minimize the lost opportunities for performance. Hand-unroll two iterations of the loop in your reordered code from Exercise 3.5.

c. [10] <3.1, 3.2> What speedup did you obtain? (For this exercise, just color the $N + 1$ iteration's instructions green to distinguish them from the Nth iteration's instructions; if you were actually unrolling the loop, you would have to reassign registers to prevent collisions between the iterations.)

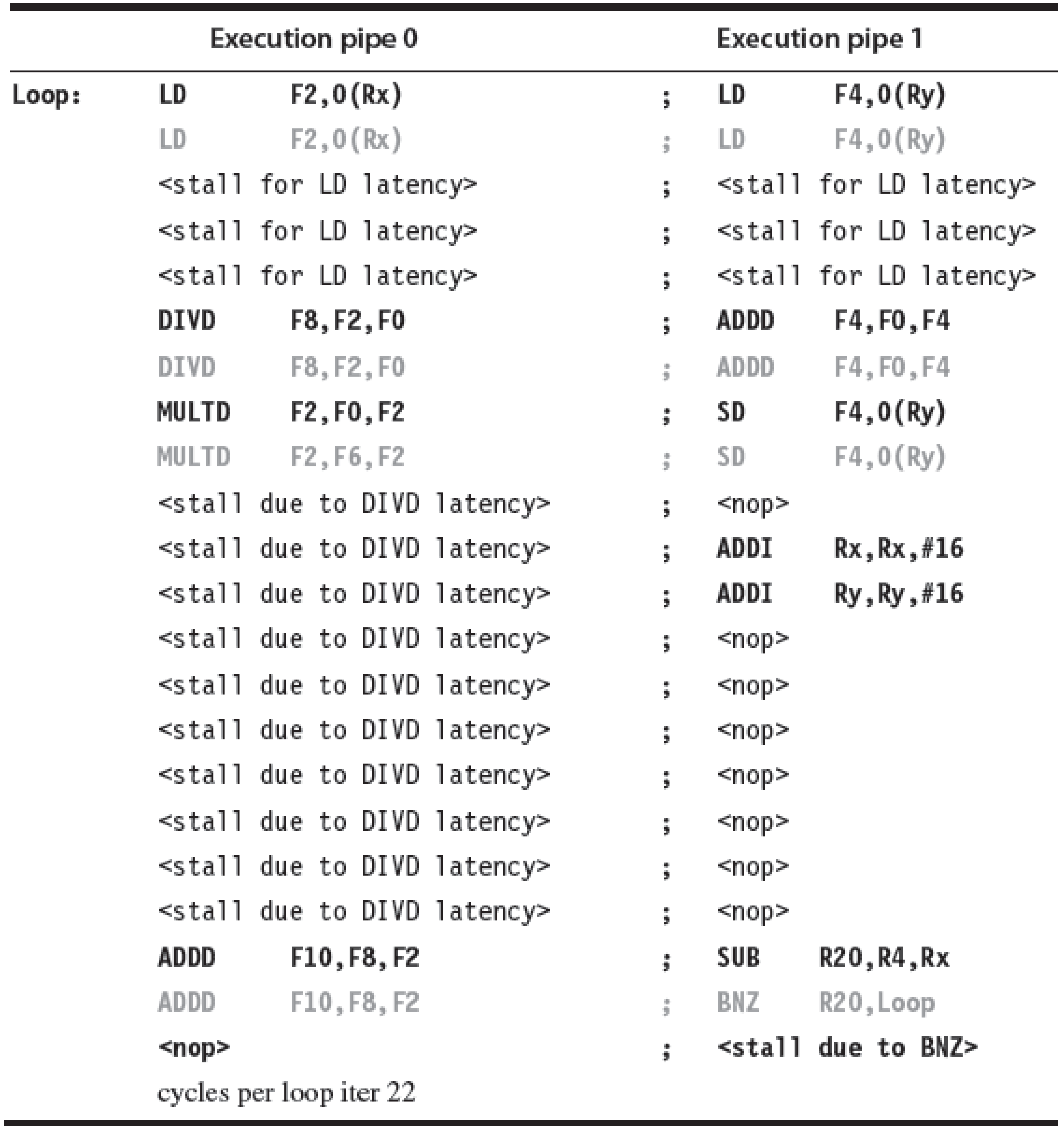
P6 a - solution

- The fraction of all cycles (for both pipes) wasted in the reordered code in Problem 5 is:

  0.275 = 11 operation / 2 x 20 opportunities ➔ 27.5%

  1- 0.275 = 0.725 ➔ 72.5 % pipeline utilization

P6b -solution



	Execution pipe 0			Execution pipe 1	
Loop:	LD	F2,0(Rx)	;	LD	F4,0(Ry)
	LD	F2,0(Rx)	;	LD	F4,0(Ry)
	<stall for LD latency>		;	<stall for LD latency>	
	<stall for LD latency>		;	<stall for LD latency>	
	<stall for LD latency>		;	<stall for LD latency>	
	DIVD	F8,F2,F0	;	ADDD	F4,F0,F4
	DIVD	F8,F2,F0	;	ADDD	F4,F0,F4
	MULTD	F2,F0,F2	;	SD	F4,0(Ry)
	MULTD	F2,F6,F2	;	SD	F4,0(Ry)
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	ADDI	Rx,Rx,#16
	<stall due to DIVD latency>		;	ADDI	Ry,Ry,#16
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	<stall due to DIVD latency>		;	<nop>	
	ADDD	F10,F8,F2	;	SUB	R20,R4,Rx
	ADDD	F10,F8,F2	;	BNZ	R20,Loop
	<nop>		;	<stall due to BNZ>	
	cycles per loop iter 22				

P6c -solution

- Speedup = Execution time *without* enhancement / Execution time *with* enhancement

S = 20 / (22/2) = 20 /11 = 1.82