Concepts Introduced in Appendix C

- basics of pipelining
- hazards
  - structural, data, control
- pipeline implementation
- exceptions
- multicycle operations

Instruction Pipelining

- Pipelining is like an assembly line.
- Each step is called a pipe step and is a machine cycle.
- Different steps from different instructions are processed in parallel.
- Pipelining improves throughput.

Pipeline Stages

- IF (Instruction Fetch): fetches the instruction from the instruction cache and increments the PC.
- ID (Instruction Decode):
  - Decode the instruction.
  - Reads two values from the register file.
  - Sign extends the immediate value.
  - Calculates the PC-relative target address of a branch.

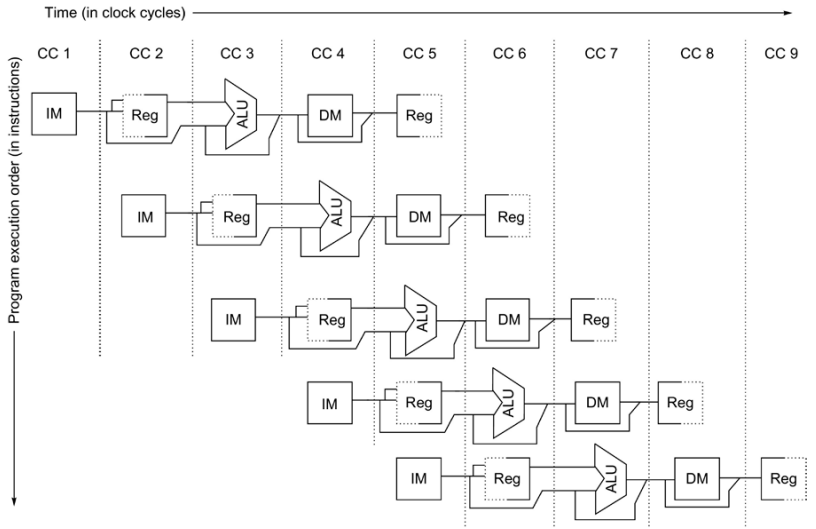
Pipeline Stages (cont.)

- EX (Execution/Effective Address):
  - Calculates an effective address for accessing memory.
  - Performs an arithmetic/logical operation on the two register values.
  - Performs an arithmetic/logical operation on a register value and the sign extended immediate value.
  - Checks if the branch should be taken.
- MEM (Memory Access): loads a value from or stores a value into the data cache.
- WB (Write Back): updates the register file with the result of an operation or a load.

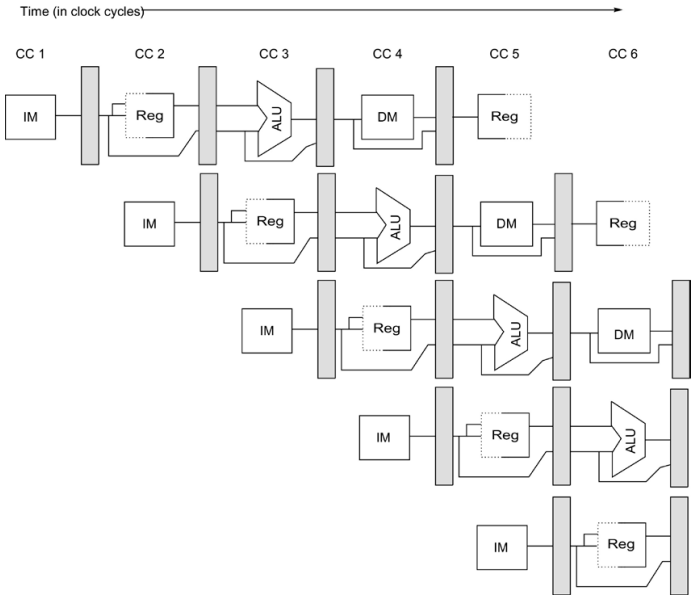
Pipline Diagram for the Simple RISC Pipeline

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Can View Each Instruction as Having Its Own Datapath



Pipeline Stages Are Separated by Pipeline Registers



Limitations of Pipeline Performance Improvements

- Ideally, performance would be improved by a factor of the number of stages in the pipeline. But there are a number of elements that limit this improvement.
  - Pipeline overhead due to passing information through the pipeline registers.
  - Imbalance between pipe stages.
  - Pipeline hazards can prevent an instruction from executing in its designated cycle.
    - structural - resource conflicts
    - data - result dependencies
    - control - changing the program counter

$$speedup = \frac{pipeline\_depth}{1+pipeline\_stall\_cycles\_per\_instruction}$$

Structural Hazards

- A structural hazard occurs when a resource is needed, but is unavailable.
- Why do structural hazards occur?
  - A functional unit may not be fully pipelined.
  - A resource may not be duplicated enough to allow certain combinations of instructions in the pipeline.
  - There is a separate L1 IC and L1 DC to avoid a structural hazard for accessing the memory system, which could occur in cycle 4 below.

```
      1   2   3   4   5   6   7   8   9
ld  IF  ID  EX  MEM WB
...     IF  ID  EX  MEM WB
...         IF  ID  EX  MEM WB
...             IF  ID  EX  MEM WB
```

- Why not design the hardware to always avoid structural hazards?
  - Some hazards infrequently occur, so the cost would outweigh the benefit.
  - If a functional unit is not fully pipelined, then it can more quickly produce its results.

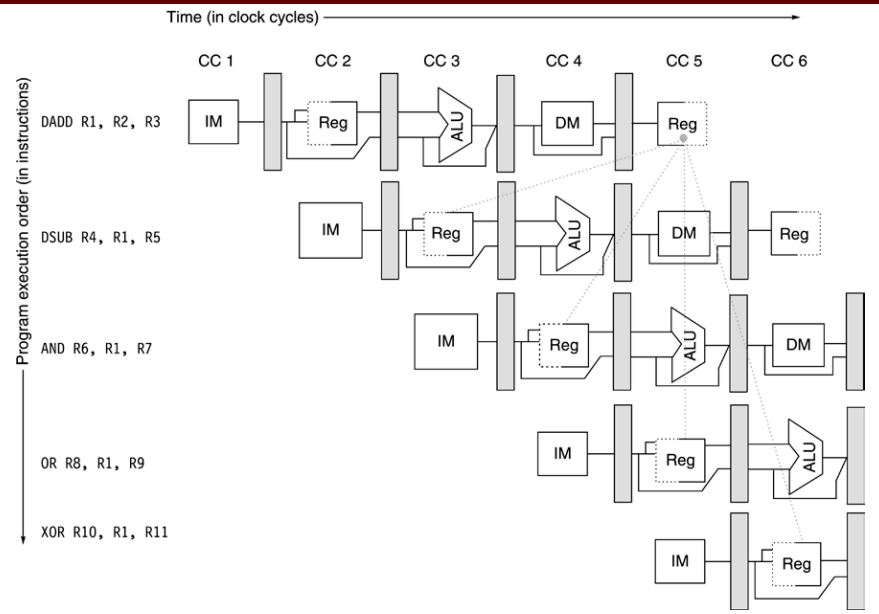
Dependences

- dependences between instructions
  - Constrain the order in which results must be calculated.
  - Indicate the possibility of hazards.
  - Set a limit on the amount of parallelism that can be exploited.
- types of dependences
  - data (true) dependences
  - name (false) dependences
  - control dependences

Data Hazards

- A data hazard arises when an instruction depends on the results of a previous instruction that is exposed in the instruction pipeline.
- types
  - RAW (read after write) - most common type of hazard
  - WAW (write after write) - Cannot occur in the integer pipeline since all writes occur in the WB stage.
  - WAR (write after read) - Cannot occur in the integer pipeline because reads occur in the ID stage and writes occur later in the WB stage.
- In the integer pipeline that is presented in the text, only loads can cause RAW stalls.

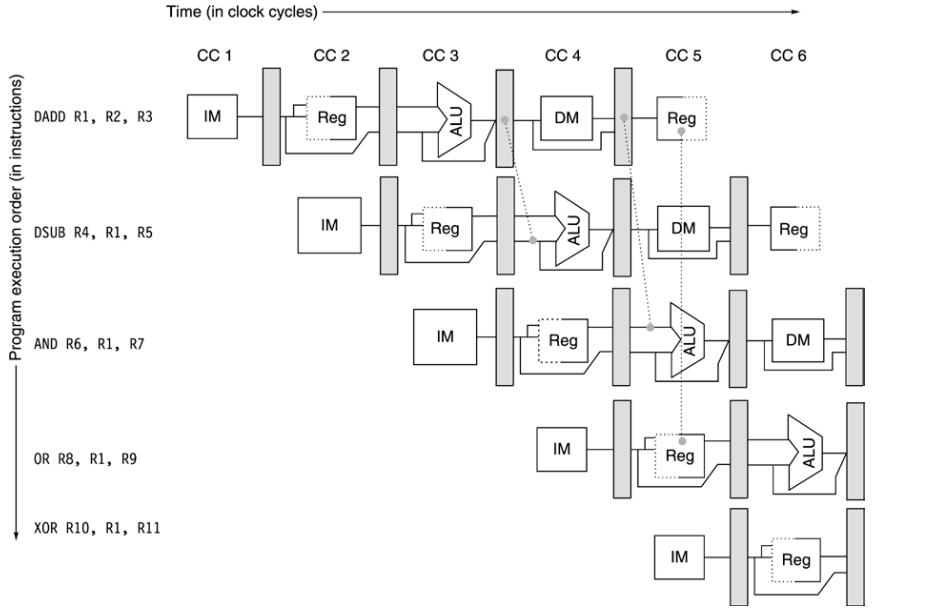
Data Hazards Example



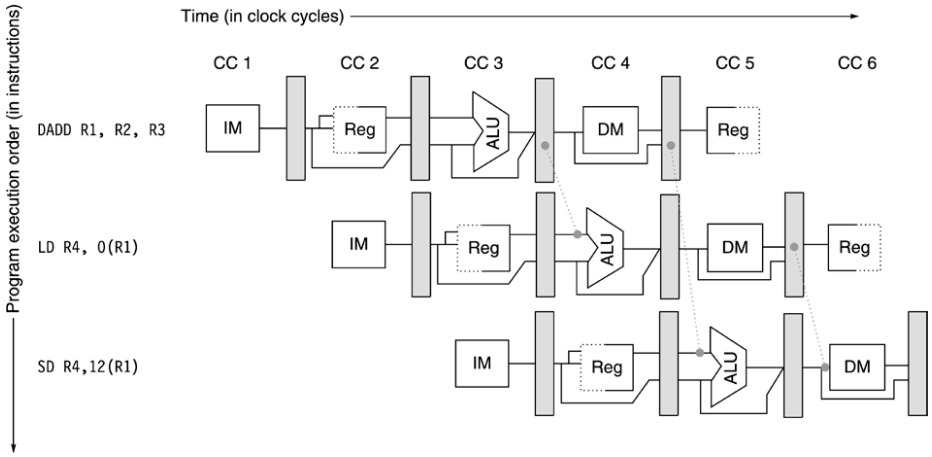
Forwarding

- Data values can be forwarded from pipeline registers (instead of the register file) when they are available.
- Data paths are set up to forward values from the EX/MEM and MEM/WB pipeline registers.
- Comparators check if the register source number for the current operation matches the register destination number for an operation that entered the pipeline earlier.
- Control logic selects the result from the register file or the forwarded value depending on whether there was a match between the destination and subsequence source register numbers.

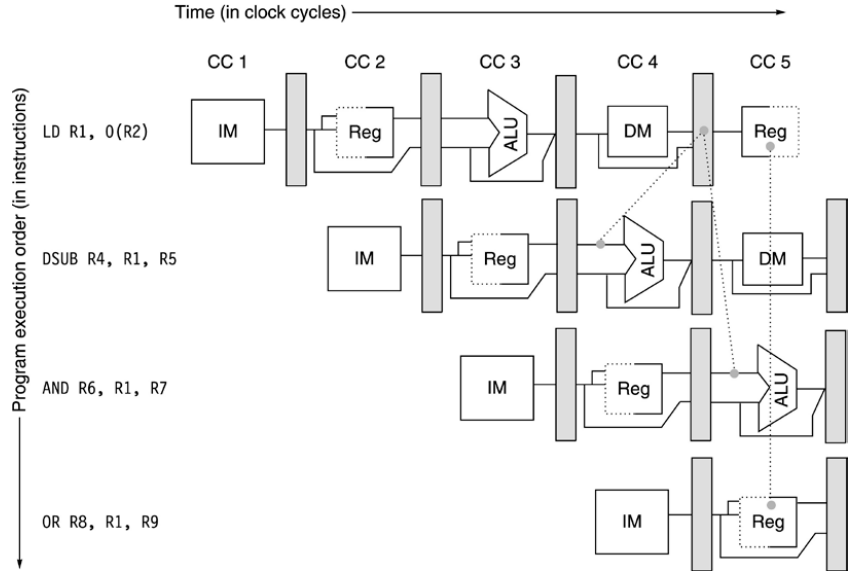
Forwarding Example



Another Forwarding Example



Load Hazard Example

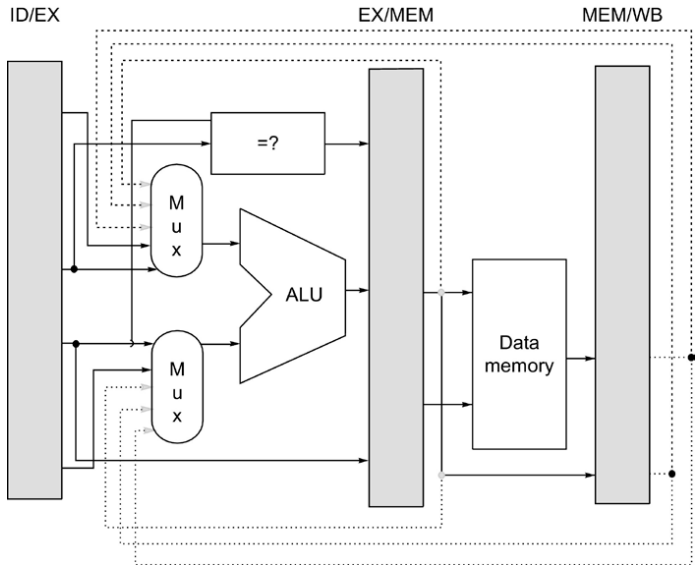


Resolving Load Hazard by Stalling Example

- The top half of the pipeline diagram will have the *sub* instruction execute with the wrong value.
- The bottom half of the pipeline diagram will have the *sub* instruction correctly execute due to the stall.

`ld x1,0(x2)`	IF	ID	EX	MEM	WB				
`sub x4,x1,x5`		IF	ID	EX	MEM	WB			
`and x6,x1,x7`			IF	ID	EX	MEM	WB		
`or x8,x1,x9`				IF	ID	EX	MEM	WB	
`ld x1,0(x2)`	IF	ID	EX	MEM	WB				
`sub x4,x1,x5`		IF	ID	Stall	EX	MEM	WB		
`and x6,x1,x7`			IF	Stall	ID	EX	MEM	WB	
`or x8,x1,x9`				Stall	IF	ID	EX	MEM	WB

Forwarding Hardware



Forwarding Examples

Situation	Example code sequence	Action

No dependence	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x6,x7`
	`or x9,x6,x7`
		No hazard possible because no dependence exists on x1 in the immediately following three instructions

Dependence requiring stall	`ld x1,45(x2)`
	`add x5,x1,x7`
	`sub x8,x6,x7`
	`or x9,x6,x7`
		Comparators detect the use of x1 in the `add` and stall the `add` (and `sub` and `or`) before the `add` begins EX

Dependence overcome by forwarding	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x1,x7`
	`or x9,x6,x7`
		Comparators detect use of x1 in `sub` and forward result of load to ALU in time for `sub` to begin EX

Dependence with accesses in order	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x6,x7`
	`or x9,x1,x7`
		No action required because the read of x1 by `or` occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half

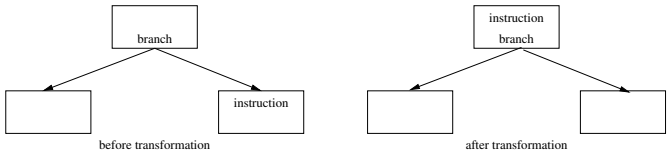
Instruction Scheduling

- Sometimes stalls due to data hazards can be avoided by reordering instructions.

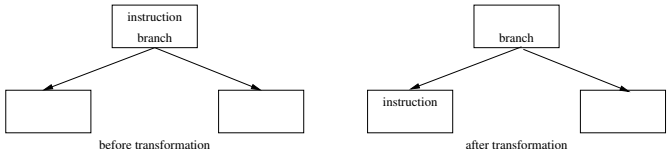
```
ld    x1,0(x4)
ld    x2,4(x4)
add   x3,x2,x1
sd    x3,8(x4)
addi  x4,x4,4

=>

ld    x1,0(x4)
ld    x2,4(x4)
addi  x4,x4,4
add   x3,x2,x1
sd    x3,4(x4)
```

Control Dependences

- An instruction is control dependent on a branch instruction if the instruction will only be executed when the branch has a specific result.
- An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is not controlled by the branch.



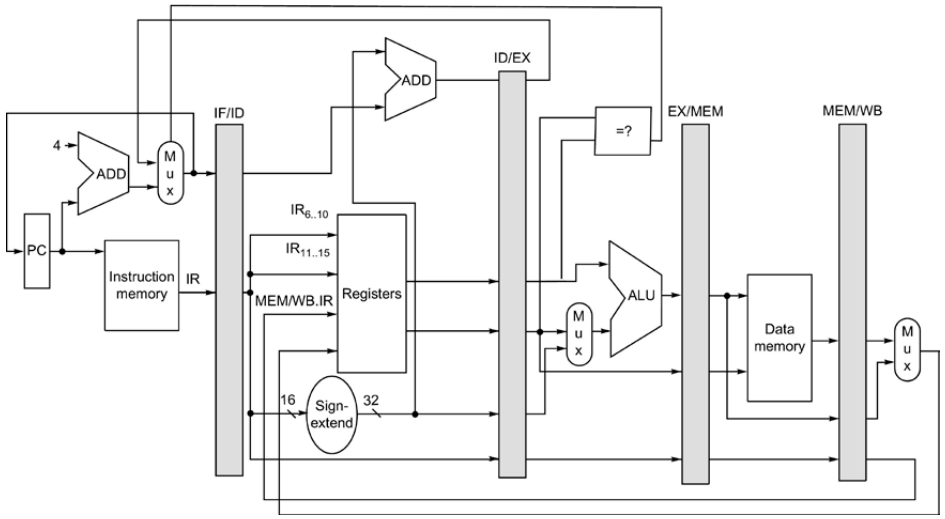
- An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.



Control Hazards

- A control hazard occurs because the CPU does not know soon enough:
  - whether or not a conditional branch will be taken and
  - the target address of the transfer of control.
- To avoid stalls the CPU can:
  - determine this information earlier or
  - delay the execution of the branch until after the information is known.

Resolving Branches Earlier



Addressing Control Hazards

- predict not taken
- predict taken
- delayed branches
- branch prediction and target buffers

Predict Not Taken Approach

- There is no delay when the branch is not taken.
- The next sequential instruction is flushed and there is a one cycle delay when the branch is taken.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

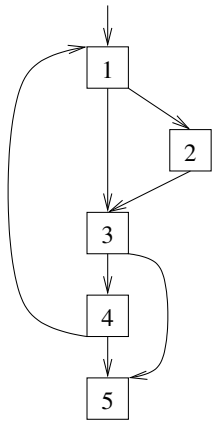
Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target+1				IF	ID	EX	MEM	WB	
Branch target+2					IF	ID	EX	MEM	WB

Types of Branch Prediction

- branch prediction can be
  - static: Predicted only by the compiler.
  - semi-static: Predicted by the compiler with the use of profile data.
  - dynamic: Predicted by the hardware at run-time.

Static and Semi-Static Branch Prediction

- Assist dynamic branch predictors for default predictions.
- Guide compiler optimizations, including instruction scheduling, code duplication for frequent paths, and aligning blocks so more branches are not taken.



Branch Prediction Buffers

- Indexed by the lower portion of the branch address.
- Contains a bit indicating if the branch was last taken or fell through.
- No tag is needed.
- Some versions have 2-bit predictors.
- Advantage is fairly good accuracy with very little memory.
- Disadvantage is that it does not help to know if a branch will be taken unless the target address has been calculated.

1-bit Branch Prediction Buffer

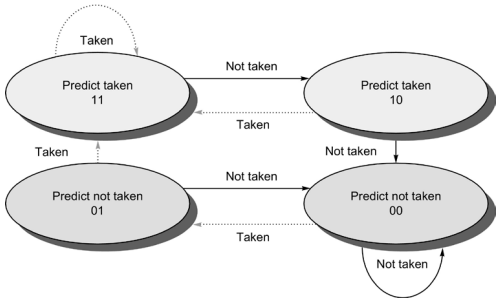
- Each element of this memory contains a single bit indicating if the branch was last taken or not.
- If the prediction turns out to be wrong, then the bit is inverted.
- Will typically mispredict a loop branch twice for each execution of the entire loop.
- How often will each of the two branches associated with the following source code be mispredicted with a 1-bit predictor?

```
for (i = 0; i < 100; i++)
   if (i & 1)
      ...
```


2-bit Branch Prediction Buffer

- Each element of this memory contains two bits indicating one of possibly four states.
- A saturating counter is used.
- A branch has to mispredict twice before the prediction is changed.
- How often will each of the following two branches now be mispredicted with a 2-bit predictor?

```
for (i = 0; i < 100; i++)
   if (i & 1)
      ...
```



Exceptions

- Force a trap instruction into the pipeline for the next IF.
- Turn off all writes for the instruction with the exception and all instructions that follow it until the trap is processed.
- The exception-handling function first always saves the PC of the instruction with the exception.
- The exception-handling function reloads this PC after handling the exception so the program can be restarted at that point.

Five Exception Attributes

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Exceptions That May Occur in the RISC V Pipeline

ld	IF	ID	EX	MEM	WB	
add		IF	ID	EX	MEM	WB

Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

Multiple Exceptions

- Example sequence of instructions.

```
ld      IF  ID  EX  MEM WB
add         IF  ID  EX  MEM WB
and             IF  ID  EX  MEM WB
sub                 IF  ID  EX  MEM WB
```

- Multiple exceptions can occur in the same cycle.
  - The *ld* could have a page fault in the MEM stage and the *add* could have an integer overflow in the EX stage.
- Exceptions can occur out of order.
  - The *and* could have a page fault in the IF stage and this would occur before the *ld* has a page fault in the MEM stage.

Supporting Precise Exceptions

- The exception addressed first is the one associated with the instruction that entered the pipeline first.
- The instructions that previously entered the pipeline are allowed to complete.
- The instruction with the exception and the ones that entered the pipeline after it are flushed.
- The appropriate instruction can be restarted after the exception is handled or the program can be terminated.

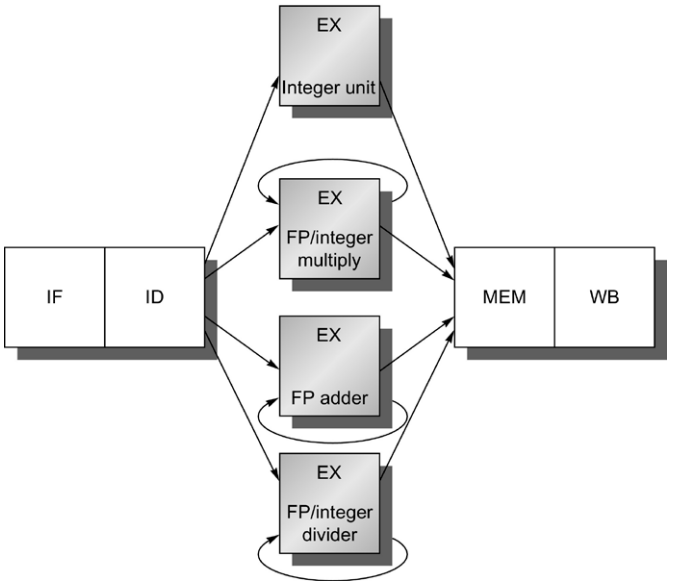
Techniques for Supporting Precise Exceptions

- Buffer the results of an operation until all of the operations that were issued earlier have completed.
- Allow out-of-order completion, but keep enough state so that the trap handling routine can complete instructions that precede the latest instruction that completed.
- Only allow an instruction to issue if instructions issued beforehand are sure to complete without causing an exception.

Multiple Cycle Operations

- Many arithmetic operations are traditionally performed in multiple cycles.
  - integer and floating-point multiplies
  - integer and floating-point divides
  - floating-point adds, subtracts, and conversions
- Completing these operations in a single cycle would require a longer clock cycle and/or much more logic in the units that perform these operations.

Three Additional Unpipelined FP Functional Units

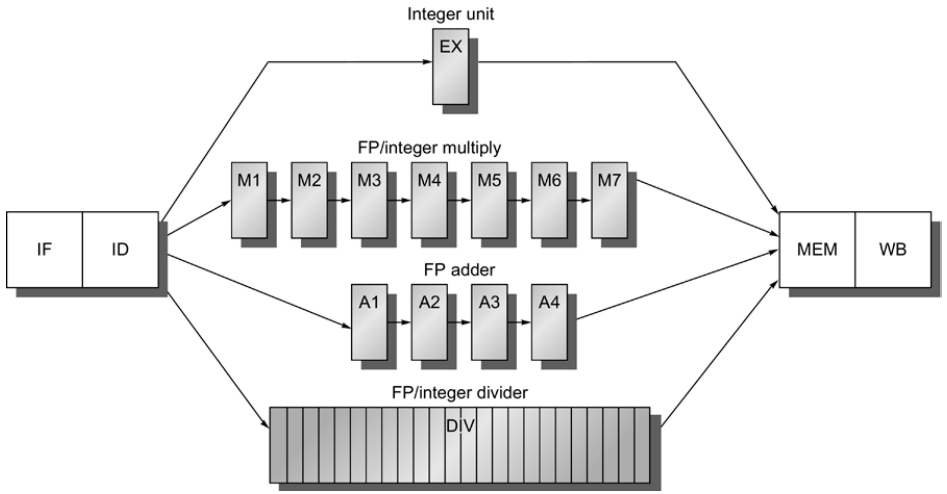


Latencies and Initiation Intervals for Functional Units

- The latency is the number of intervening cycles between an instruction that produces a result and an instruction that uses the result.
- The initiation interval is the number of cycles that must elapse between issuing two operations of a given type.

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Pipeline Supporting Multiple Outstanding FP Operations



Pipeline Example with a Set of Independent FP Operations

- The results below are completed out of order.

fmul.d	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	FWB
fadd.d		IF	ID	*A1*	A2	A3	**A4**	FWB		
fld			IF	ID	*EX*	**MEM**	FWB			
fsd				IF	ID	*EX*	**MEM**			

Complications Due to Multiple Cycle Operations

- Structural hazards can occur when multicycle operations are not fully pipelined.
- Multiple instructions can attempt to write to the FP register file in a single cycle.
- WAW hazards are possible since instructions may not reach the WB stage in order.
- Out of order completion may cause problems with exceptions.
- Stalls for RAW hazards will be more frequent.
- The longer the pipeline, the more complicated the stall and forwarding logic becomes.

Pipeline Example with Dependent FP Operations

- Longer latency operations result in more stall cycles.

	Clock Cycle Number															
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`fld f4,0(x2)`	IF	ID	EX	MEM	WB											
`fmul.d f0,f4,f6`		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	FWB				
`fadd.d f2,f0,f8`			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	FWB
`fsd f2,0(x2)`					IF	stall	stall	stall	stall	stall	stall	ID	stall	stall	EX	MEM

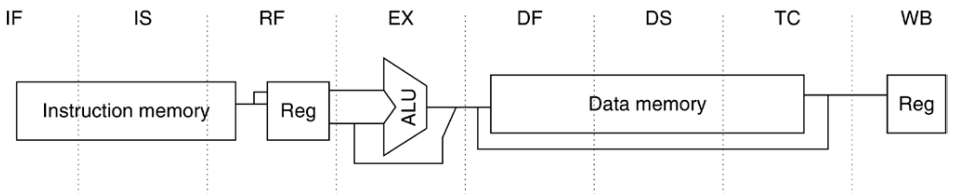
Pipeline Example with WB Structural Hazards

- FP operations with different latencies can lead to WB structural hazards.
- Completing instructions out of order can lead to WAW hazards and imprecise exceptions.

	Clock Cycle Number									
Instruction	1	2	3	4	5	6	7	8	9	10
`fmul.d f0,f4,f6`	IF	ID	M1	M2	M3	M4	M5	M6	M7	FWB
...		IF	ID	EX	MEM	WB				
...			IF	ID	EX	MEM	WB			
`fadd.d f2,f4,f6`				IF	ID	A1	A2	A3	A4	FWB
...					IF	ID	EX	MEM	WB	
...						IF	ID	EX	MEM	WB

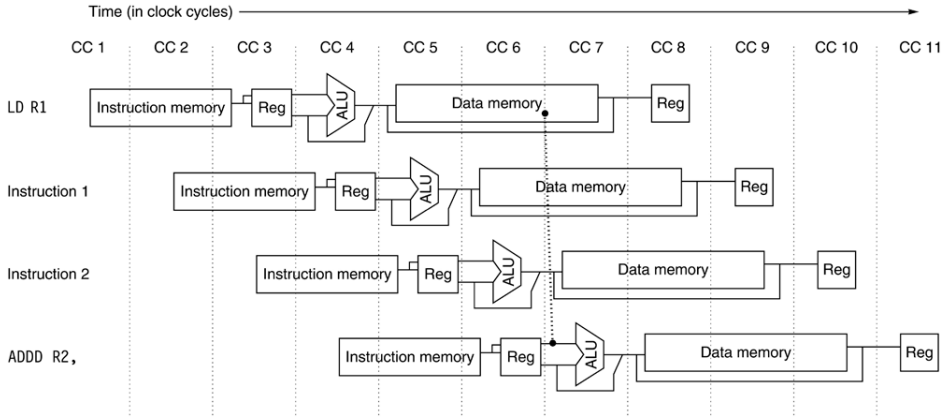
MIPS R4000 8-Stage Pipeline

- Cache accesses often take multiple cycles and are pipelined in today's processors.



Multicycle Cache Accesses Leads to a Longer Load Delay

- Instruction scheduling is important to avoid load hazards on a processor with a multicycle data cache access.



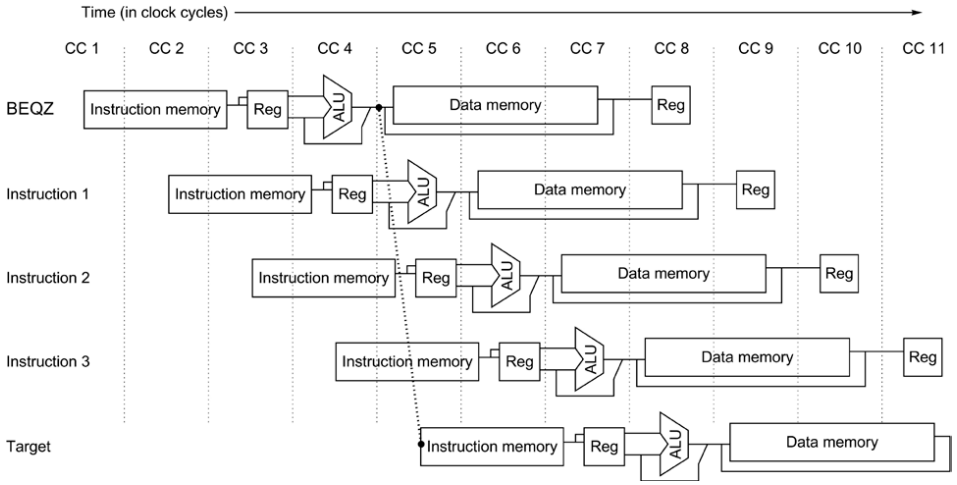
Multicycle Cache Access Leads to More Load Hazards

- Multicycle data cache access can often lead to load hazard stalls.

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`ld x1,...`	IF	IS	RF	EX	DF	DS	TC	WB	
`add x2,x1,...`		IF	IS	RF	Stall	Stall	EX	DF	DS
`sub x3,x1,...`			IF	IS	Stall	Stall	RF	EX	DF
`or x4,x1,...`				IF	Stall	Stall	IS	RF	EX

More Pipeline Stages Can Lead to More Branch Delays

- The delay for a taken branch is 3 cycles in the MIPS R4000.



MIPS R4000 8 FP Pipeline Stages

- FP operations consist of several steps.

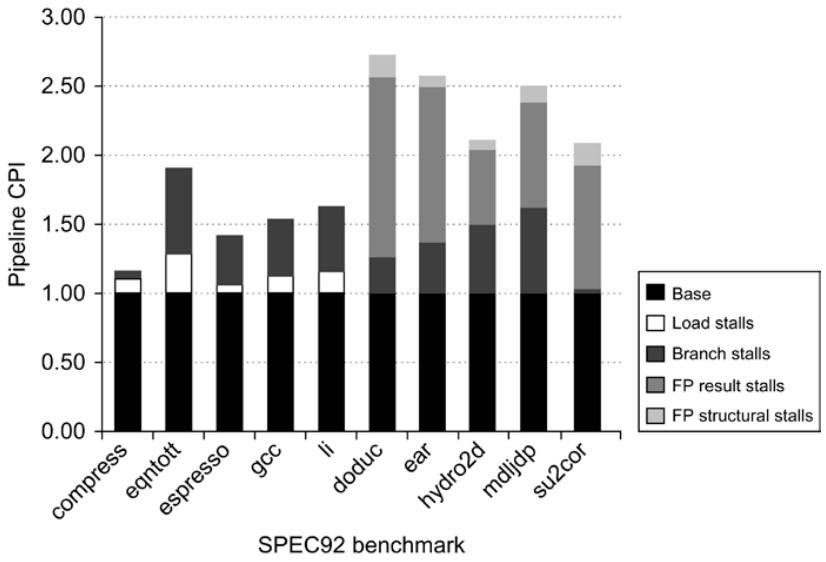
Stage	Functional unit	Description
A	FP adder	Mantissa add stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

MIPS R4000 FP Latencies and Initiation Intervals

- When the initiation interval is one less than the latency, then the operation is generally not pipelined.

FP instruction	Latency	Initiation interval	Pipe stages
Add, subtract	4	3	U, S+A, A+R, R+S
Multiply	8	4	U, E+M, M, M, M, N, N+A, R
Divide	36	35	U, A, R, D^{28}, D+A, D+R, D+A, D+R, A, R
Square root	112	111	U, E, $(A+R)^{108}$, A, R
Negate	2	1	U, S
Absolute value	2	1	U, S
FP compare	3	2	U, A, R

Pipeline CPI for 10 SPEC92 Benchmarks



Fallacies and Pitfalls

- Pitfall: Unexpected execution sequences may cause unexpected hazards.

```
        bne     x1,x0,foo
        div.d   f0,f2,f4      # in delay slot
        ...                   # from fall through
foo:    fld     f0,qrs
```

- Pitfall: Extensive pipelining can impact other aspects of a design, leading to overall worst cost-performance.