

Branch behavior

- Observations using SPEC subset on DLX:
 - Dynamic frequencies (used here) count number of executions
 - Static frequencies count number of occurrences in program
 - Conditional branches are much more common:
- Conditional outnumber unconditional about 3-4 to 1.
 - 14% to 16% is normal for integer benchmarks
 - FP benchmarks are much more varied at 3%-12%
- Forward branches more common: outnumber backwards branches by 3 to 1
- Frequency of taken branches
 - 67% of conditional branches are taken on average.
 - 60% of the forward and 85% of the backward branches.

Reducing branch penalties

- Move the address calculation and decision of whether to take the branch back into ID
 - If the comparison is to zero, we know the address and the decision at the end of ID.
 - If comparing one register to another, we wait until after EX to decide if the branch is taken
- Method 1: freeze pipeline until decision known
 - Always flush the pipeline of instructions up until the branch destination and condition are known
 - Branch penalty is fixed and cannot be reduced by software & compiler

Static branch prediction

- Treat every branch as not taken (predict-not-taken)
 - Continue to fetch instructions.
 - Flush the pipeline if the branch is taken.
 - Note that successor instructions may NOT change the state of the machine

⇒ This results in a 1 cycle stall for DLX since the decision (for zero compares) is known after ID
- Treat every branch as taken (predict-taken)
 - Wait until the target address is computed and then fetch instructions using the new PC value.
 - Flush the pipeline if the branch is NOT taken.
 - For DLX, this doesn't do much good since BOTH the branch target address and the decision (for zero compares) is known after ID

Delayed branches

- An execution cycle with a branch delay of length n is:

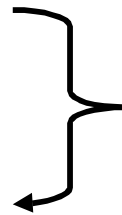
Branch instruction

Sequential successor₁

...

Sequential successor _{n}

Branch target



Branch delay slots

- Instructions(s) in the branch delay slot(s) after the branch are always executed
 - The compiler should try to put a “useful” instruction here.
 - If none are available, then a “no-op” is inserted in the delay slot.
- Improves performance by letting the CPU do useful work while waiting for branch target and condition resolution

Delayed branch scheduling

From before

```
ADD R1,R2,R3
BEQZ R2,label
delay slot
```



```
BEQZ R2, label
ADD R1,R2,R3
```

From target

```
loop:
SUB R4,R5,R6
ADD R1,R2,R3
BEQZ R1,loop
delay slot
```



```
SUB R4,R5,R6
loop:
ADD R1,R2,R3
BEQZ R2,loop
SUB R4,R5,R6
```

From fall-through

```
ADD R1,R2,R3
BEQZ R1,label
delay slot
label:
SUB R4,R5,R6
```



```
ADD R1,R2,R3
BEQZ R1,label
SUB R4,R5,R6
label:
```

Delayed branch limitations

- Restrictions on the instructions that are scheduled into the delay slots (i.e. data dependencies.)
 - Compiler's ability to predict accurately whether or not a branch is taken determines how much useful work is actually done.
- Many machines have introduced a cancelling or nullifying branch instruction.
 - Includes the direction that the branch is predicted to go.
 - If branch is predicted incorrectly, CPU turns the instruction in the branch delay slot into a no-op.
- Compilers can usually fill a single branch delay slot & improve performance
- More delay slots => more difficult to fill with useful work

Delayed branch performance

Pipeline performance: $\text{pipeline speedup} = \frac{\text{pipeline depth}}{1 + \text{pipeline stalls from branches}}$

This assumes no delays from other hazards...

Calculate pipeline stalls due to branches by:

$\text{stall cycles from branches} = \text{branch frequency} \times \text{branch penalty}$

Total pipeline speedup is thus:

$\text{pipeline speedup} = \frac{\text{pipeline depth}}{1 + \text{branch frequency} \times \text{branch penalty}}$

Branch performance example

- Assume: branches have a single delay slot
 - Filled with a useful instruction 65% of the time
- Assume: branch condition not known for two cycles beyond the delay slot
 - If predicted properly, there is no penalty
 - If mispredicted, the two intervening instructions must be cancelled
- Forward branches (75%) are always predicted not taken
- Backward branches (25%) are always predicted taken
- Branches are 20% of all instructions.
 - 50% of forward branches taken
 - 85% of backward branches are taken
- What is the new CPI (assuming the original CPI is 1)?

Branch performance example: solution

- For 35% of the branch instruction, the delay slot isn't filled
 - This adds 0.35 cycles of branch stalls
- 50% of forward branches suffer a 2 cycle penalty.
 - 75% of branches are forward $\Rightarrow 0.50 * 0.75 * 2 = 0.75$ cycles
- 15% of backward branches suffer a 2 cycle penalty
 - Penalty is $0.15 * 0.25 * 2 = 0.075$ cycles
- Total branch penalty is $0.35 + 0.75 + 0.075 = 1.175$ cycles.
- Since branches make up 20% of all instructions, the penalty to the CPI is $1.175 * 0.2 = 0.235$ cycles.
- The new CPI is thus $1 + 0.235 = 1.235$

More compiler optimizations for branches

- Having accurate information about branch behavior at compile time is also helpful for scheduling data hazards
- Suppose we knew that the branch was almost always taken and value in R7 was not needed in the fall through part
 - Compiler could move ADD R7, R8, R9 after the load instruction
- Suppose we knew that the branch was rarely taken and value in R4 was not needed on the taken path
 - Compiler could move OR R4, R5, R6 after the load instruction.
- These optimizations are in addition to any branch delay scheduling.

```
LW R1, 0(R2)
```

```
SUB R1, R1, R3
```

```
BEQZ R1, L
```

```
OR R4, R5, R6
```

```
ADD R10, R4, R3
```

```
L: ADD R7, R8, R9
```

Compilers & static branch prediction

- Predict all branches taken
 - Surprisingly effective since 85% of backward branches and 60% of forward branches are taken
 - Still leaves more than a third of the branches improperly predicted
 - For some programs, this method is excellent (< 10% mispredictions), but for others, it does badly (> 50%)
- Predict forward not taken and backward taken
 - This scheme is similar to predicting all branches as taken except that it uses information about the types of branches.
 - Forward branches are usually part of if-else constructs, and may be less likely to be taken
 - Backward branches are often part of loops and more likely to be taken
 - Won't perform much better than simply predicting not-taken

Using profiling to predict branches

- Use profile information from previous runs
 - The compiler can instrument the code using the profile information from previous runs of the program.
 - It can build a higher performance program by predicting that branches taken in the practice run(s) will be taken in the final version.
- Not perfect since many branches are both taken and not taken in the course of execution.
 - Provides better prediction than other static methods.
 - Misprediction rates for this method range from 5% to 20%, even if different input data is used for the program

Wrapping up static branch techniques

- Studies have shown that profile-based prediction is almost always better than predict-taken or other non-profile-based methods
 - Since profile-based prediction is so good, why not use it?
 - Limits to static prediction
 - Branches behave differently at different times
 - Behavior can be very input-dependent
- Dynamic branch prediction provides a better solution
 - Predict branch behavior while program is running
 - Visit this topic in a week or two

Pipeline difficulties

- Why is pipelining difficult?
 - Now that we've seen how pipelining can be done and how to detect and resolve hazards, the question arises: what's so hard about this?
 - Exceptions
 - Instruction set complications
- Exceptions
 - An instruction in the pipeline can raise an exception that may force other instructions in the pipeline to be aborted
 - These other instructions may have altered the state of the machine.
 - Exceptions introduce the possibility that an exception in a later instruction (i.e. in ID or EX) will prevent a previous instruction (i.e. in MEM or WB) from completing

Exception causes

- I/O device requests
- User OS service requests
- Breakpoints
- Integer arithmetic overflow/underflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory accesses
- Memory protection violations
- Hardware malfunctions
- Undefined instructions

Classifying exceptions

- Synchronous vs. asynchronous
 - Synchronous: exception comes as a result of execution at the same place for every run of a program with the same data and memory allocation
 - Asynchronous: generated external to the CPU
 - Asynchronous events can usually be handled after the completion of the current instruction, making them easier to handle.
- User requested vs. coerced
 - Did the user request an exception, i.e. through a trap?
 - Or did it happen as a result of something beyond the user program's control, i.e. a hardware event?
 - Coerced exceptions are harder to implement since they are not predictable

Classifying exceptions

- User maskable vs. non-maskable
 - Can the user prevent the hardware from responding?
 - Note that for maskable interrupts, the user can choose to respond to them, and therefore they are similar to non-maskable interrupts
 - Maskable interrupts must still be handled properly!
- Within vs. between instructions
 - Does the exception prevent instruction completion, by occurring in the middle of execution?
 - Or is it recognized between instructions?
 - Exceptions occurring within instructions are usually synchronous, since the instruction triggers the exception
 - **Within** is more difficult to implement than **between** since the former must be restarted

Classifying exceptions

- Resume vs. terminate
 - Terminate: the exception stops the program from running
 - Resumable: the program must be restartable after the interrupt
 - Restarting is harder (obviously), and is the more common case
 - The most difficult case is handling interrupts within an instruction, where the instruction must be resumed
 - Save the state of the executing program
 - Fix the cause of the exception
 - Restore the state of the original program, and restart it as if nothing had happened
 - Exceptions of this type occur for virtual memory management systems

⇒ Restartable instructions

Saving pipeline state

- For exceptions that occur within instructions (i.e. in EX or MEM) and must be restarted (page fault), the pipeline state must be saved
 - Insert a trap instruction into the pipeline on the next IF.
 - Turn off all writes for the faulting instruction and the instructions following it in the pipeline
 - Allow previous instructions to complete
 - Save the PC of the faulting instruction so it can be restarted (usually done by OS)
- This method requires as many PCs as there are delay slots
 - Instructions currently in the pipeline may not be sequentially related!
 - Save at least one PC value: the location of the faulting instruction

Precise vs. imprecise exceptions

- Precise exceptions mean:
 - All instructions before the faulting instruction complete
 - Instructions following the faulting instruction, including the faulting instruction, do not change the state of the machine.
- Restarting is easy with precise exceptions!
 - Simply re-execute the original faulting instruction
 - If it's not a resumable instruction, i.e. an integer overflow, start with the next instruction
- Precise exceptions can be difficult because of instruction completions and exceptions that occur out of order
 - Solution: imprecise exceptions.
 - Often used for floating point pipelines more so than integer pipelines
- Integer -> precise, FP -> imprecise (usually)

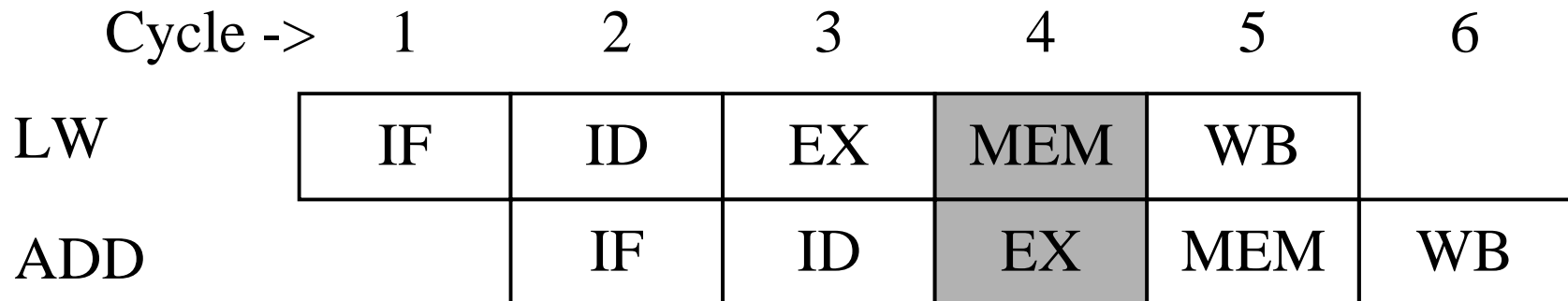
When do exceptions occur?

- IF
 - Page fault for instruction
 - Unaligned memory access
 - Memory protection fault
- ID
 - Undefined / illegal opcode
- EX
 - Arithmetic exception
- MEM
 - Page fault for data
 - Unaligned memory access
 - Memory protection fault
- WB: no faults

Exception ordering

- Two consecutive instructions cause exceptions in the same cycle
 - Which should be handled?
 - Handle the one belonging to the earlier instruction
- Cancel the later instruction (the ADD)
- Handle the page fault in the earlier (LW) instruction

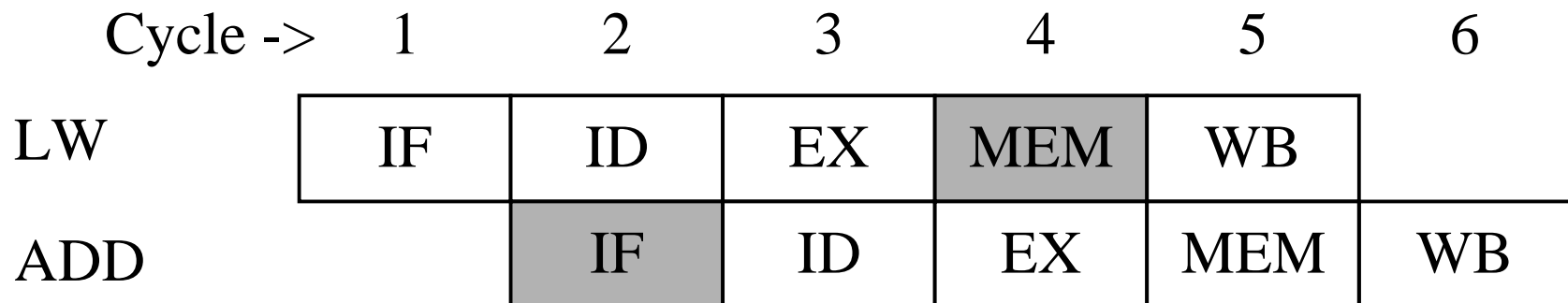
```
LW  R4, 8(R5)      ; causes page fault for data
ADD R9, R10, R11   ; causes overflow
```



Exception ordering

- Page fault in ADD occurs **first**
- Must finish LW before handling the page fault in ADD if we want precise exceptions!
- Solution: keep an **exception vector**
 - Posted exceptions added to the vector, disabling writes for that instruction
 - Vector checked at end of each instruction...

```
LW  R4, 8(R5)      ; causes page fault for data
ADD R9, R10, R11   ; causes page fault for instr
```



Exception vectors

- When the instruction is about to exit the pipeline (MEM/WB), any pending exceptions for the instruction are examined
 - If an instruction generates multiple exceptions, the exception occurring in the earliest stage takes precedence.
 - For the DLX, the faulting instruction has not updated any state (since all updates occur in WB)
- Many CPUs support both precise & imprecise exceptions
 - Precise exceptions are slower (often)
 - Imprecise exceptions are faster but don't note where the exception occurred (OK if the process will be killed anyway...)

Instruction set complications

- An instruction is **committed** when it is guaranteed to complete
 - In DLX, all instructions are committed at the end of MEM
 - Since no updates occur before instructions commit, precise interrupts are straightforward.
- In most RISC systems, each instruction writes only one result
 - Instruction can be cancelled any time before the instruction is committed, with no harm to the system state
 - Not true for many CISC machines, i.e. VAX: system state may be modified well before the instruction or its predecessors are committed
 - Example: instruction using autoincrement mode is aborted because of an exception, altering the machine state
 - It's difficult to restart the instruction or keep precise exception

Exceptions in CISC architectures

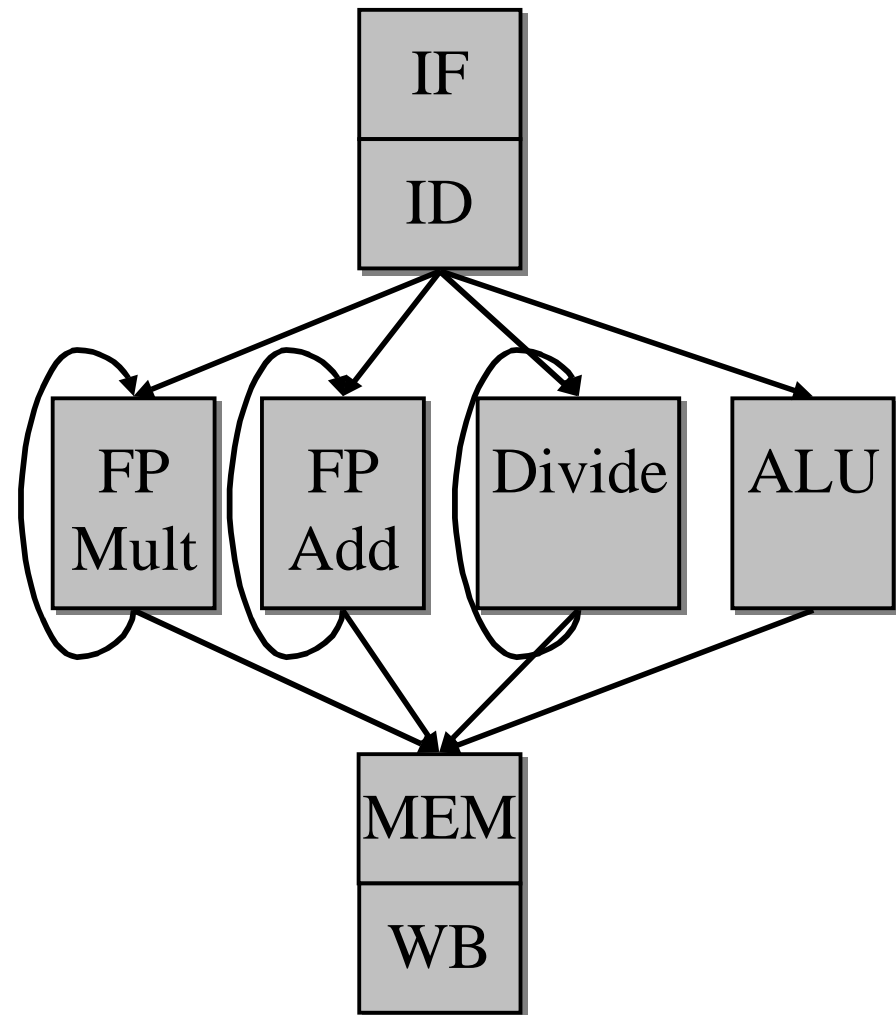
- The situation is worse for instructions that access and write memory in multiple places
 - These instructions can generate multiple faults.
 - Therefore, it becomes difficult to know where to resume
 - For string instructions, the CPU must also know how far into the operation it was when the exception occurred
 - Usually solved by using general purpose registers as scratch space (that are saved and restored)
 - General solution used by more complex instruction set machines is to pipeline the microcode.
- ⇒ **RISC has often been compared to having the microcode as the actual assembly language**

Multi-cycle pipeline operations

- Many operations can't finish in one or two cycles
 - FP operations
 - Vector operations
- It might be possible to allow it, but would require
 - Slow clock
 - Lots of logic (more than we want to dedicate...)
- Use FP operations as an example (vectors later...)
- Implement several FP units (adder, multiplier, divider, etc.)
 - Allow a longer latency (several clocks)
 - May pipeline them to avoid structural hazards
 - One possible way to implement an FP pipeline is to allow the EX stage to repeat as many times as necessary

Non-pipeline FP in DLX

- FP multiplier, adder
- FP/integer divide unit
- FP units take multiple cycles
 - Non-pipelined
 - Structural hazards may occur if successive instructions use the same functional unit
- Structural hazards can cause stalls!



Pipelining the FP functional units

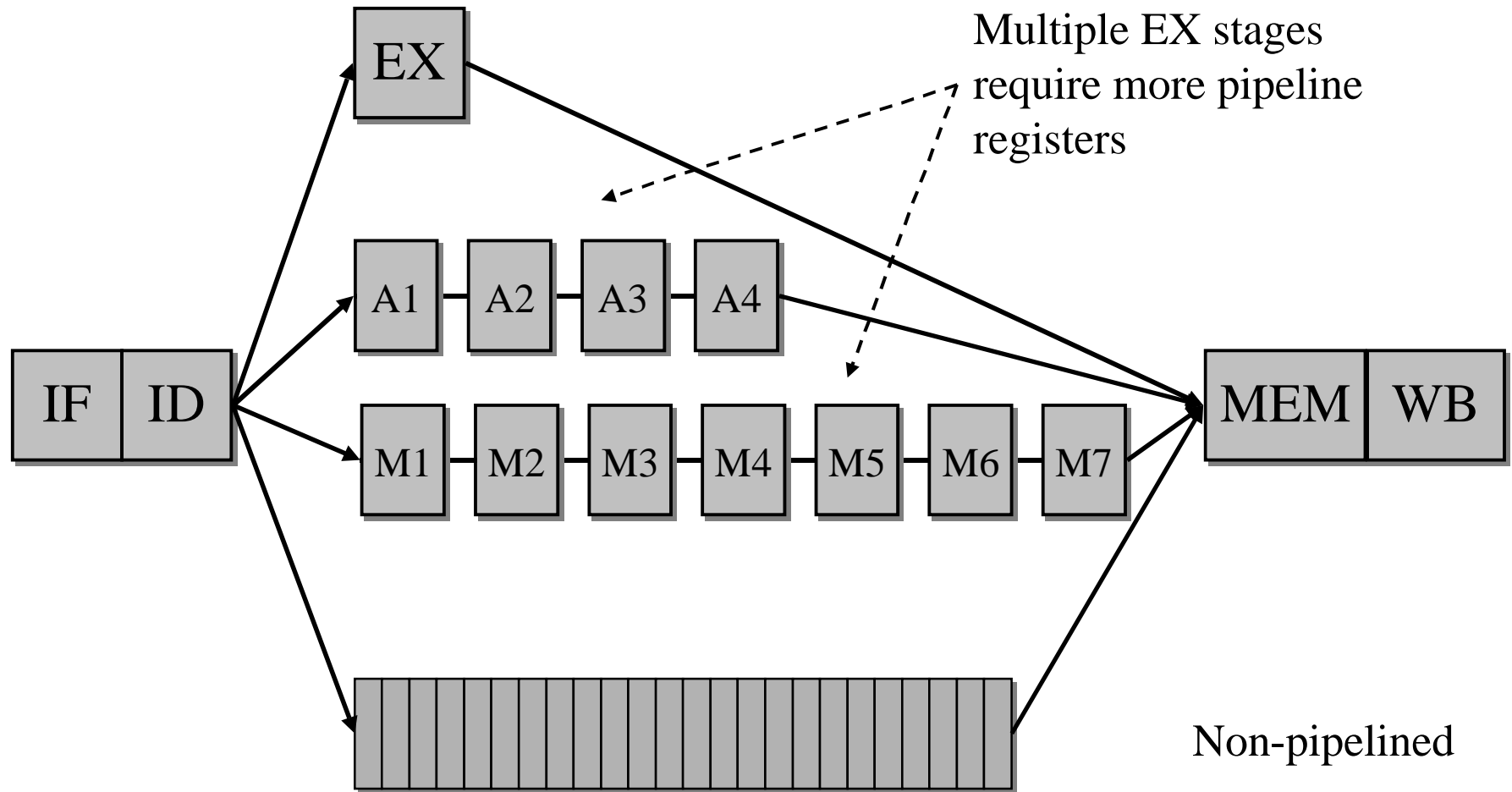
- Define the latency and the initiation interval for FP units
 - Latency => number of cycles needed beyond the first
 - Integer ALU has its result ready at the end of its first cycle
 - Integer ALU requires 0 *additional* EX stages
 - Initiation interval => number of cycles that must elapse between instruction issue to the same unit
- Pipelined units have
 - Various latencies
 - Initiation interval of 1
 - May issue a new operation every cycle
 - May have multiple outstanding operations!

Sample pipeline unit parameters

Functional unit	Latency	Initiation interval	# of pipe stages
Integer ALU	0	1	1
Data memory	1	1	1
FP add	3	1	4
FP/int multiply	6	1	7
FP/int divide	24	25	1

- Integer ALU: 0 cycle latency - result can be used on next clock cycle
- Data memory: 1 cycle latency, pipelined - result can be used after one intervening cycle
- FP add: 3 cycle latency, pipelined
- FP/integer multiplier: 6 cycle latency, pipelined
- FP/integer divide and FP square root: 24 cycle latency, non-pipelined

Pipeline including FP units



Pipeline characteristics

- Structural hazards can occur for the divide unit
- Several instructions can reach WB in a single cycle because of variable length instructions
 - More than 1 register write could occur in one cycle
 - ⇒ Structural hazard if the CPU can only write one register per cycle
- Instructions no longer necessarily complete in the order in which they were issued (out-of-order completion)
 - **WAW** hazards are now possible and must be detected.
 - More problems with exceptions: the exception handling mechanism relied on the fact that instructions completed in issue order
- Since instructions have longer latencies, stalls for **RAW** operations will be more frequent

RAW hazards in the FP pipeline

- MULTD stalls due to load latency.
- ADDD stalls until multiply produces F0 value, which is forwarded.
- SD stalls in MEM waiting on result from ADDD

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD	F4, 0(R2)	IF	ID	EX	MM	WB												
MULTD	F0, F4, F6		IF	ID	-	M1	M2	M3	M4	M5	M6	M7	MM	WB				
ADDD	F2, F0, F8			IF	-	ID	-	-	-	-	-	-	A1	A2	A3	A4	MM	WB
SD	0(R2), F2					IF	-	-	-	-	-	-	ID	EX	-	-	-	MM

Contention for the register write port

- Assume the FP register file has only one write port
- In this example, three instructions write to it in one cycle: hazard!
- Possible solutions
 - Increase the number of write ports: may not be worth it if the situation doesn't happen very often
 - Serialize the writes (stall instructions conflicting for resource)

		1	2	3	4	5	6	7	8	9	10	11
MULTD	F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MM	WB
ADD	R1, R2, #1		IF	ID	EX	MM	WB					
SUB	R4, R4, #4			IF	ID	EX	MM	WB				
ADDD	F2, F0, F8				IF	ID	A1	A2	A3	A4	MM	WB
OR	R8, R8, #8					IF	ID	EX	MM	WB		
OR	R9, R9, #1						IF	ID	EX	MM	WB	
LD	F8, 0(R7)							IF	ID	EX	MM	WB

Serializing writes to the FP registers

- Solving the write port structural hazard through serialization can be done in two ways
- Stall the instruction when it tries to enter the MEM or WB stage.
 - + Easy to detect the conflict at this point
 - Complicates pipeline control since stalls can now occur in two places
- Keep track of when each instruction will use the WB stage and stall instructions in ID if their “slot” is already in use
 - Can be done using a shift register that tracks when already-issued instructions will use the register file
 - + Instructions are stalled only in ID
 - Requires additional hardware (shift register and write conflict logic)

WAW hazards

- Consider situation below: WAW hazard since LD writes F0 one cycle earlier than MULTD
 - ONLY a hazard when MULTD is overwritten without any instruction ever using it -- it appears to be a useless instruction
 - If there was a use between MULTD and LD, then a RAW hazard would stall the pipeline and the WAW would not occur
- However, we must still detect them since they do occur in reasonable code (as we will see).

		1	2	3	4	5	6	7	8	9	10	11
MULTD	F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MM	WB
ADD	R1, R2, #1		IF	ID	EX	MM	WB					
SUB	R4, R4, #4			IF	ID	EX	MM	WB				
OR	R8, R8, #8				IF	ID	EX	MM	WB			
LD	F0, 0(R7)					IF	ID	EX	MM	WB		

Dealing with WAW hazards

- The WAW hazard can be handled in one of two ways
 - Stall an instruction that would “pass” another until after the earlier instruction reaches the MEM phase
 - Cancel the WB phase of the earlier instruction
- Either can be done in ID, i.e. when LD is about to issue
- Pure WAW hazards are not common => use either method
 - Pick the one that simplest to implement.

⇒ Simplest solution for the DLX pipeline is to hold the instruction in ID if it writes the same register as an instruction already issued

Hazard checking during ID

- Possible sources of hazards
 - Hazards among FP instructions (already handled)
 - Hazards between an FP instruction and an integer instruction
- Separate FP & integer register files => only FP loads and stores and FP register moves to integer registers involve hazards
- Assume all hazards are detected during ID
- Three checks are required in ID (before instruction issue)
 - Check for structural hazards
 - Divide unit
 - Register write port

Hazard checking during ID

- Check for RAW hazards: CPU stalls the instruction at ID stage until
 - Its source registers are no longer listed as destinations in any of the execution pipeline registers (registers between stages of M and A) OR
 - Its source registers are no longer listed as the destination of a load in the EX/MEM register
- Check for WAW hazards
 - Check instructions in A1, ..., A4, Divide, or M1, ...,M7 for the same destination register (check pipeline registers)
 - Stall instruction in ID if necessary
- Instructions after the current one might have been able to execute, but they'll all have to wait (until the next chapter...)

FP pipelining & exception handling

- Exceptions are difficult because instructions may now finish out of order

```
DIVF F0, F2, F4
```

```
ADDF F10, F10, F8
```

```
SUBF F12, F12, F14
```

- ADDF and SUBF are expected to complete before DIVF
- Suppose SUBF causes an arithmetic exception at a point where ADDF has completed but DIVF has not
 - Imprecise exception!
 - Fix here is to let pipeline drain
- Suppose DIVF has an exception after ADDF completes
 - ADDF destroys one of its operands => can not restore the state to what it was before the DIVF instruction, even with software!

Handling exceptions

- Ignore the problem (imprecise exceptions)!
 - This may be fast and easy, but it's difficult to debug programs without precise exceptions
 - Many modern CPUs,, provide a precise mode that allows only a single outstanding FP instruction at any time (DEC Alpha 21064, IBM Power-1, MIPS R8000)
 - Precise mode is much slower than the imprecise mode!
- Buffer the results and delay commitment: CPU doesn't actually make any state (register or memory) changes until the instruction is guaranteed to finish
 - Becomes difficult when the difference in running time among operations is large.
 - Lots of intermediate results have to be buffered (and forwarded...)

Handling exceptions: save values

- History file: saves the original values of the registers that have been changed recently
 - If an exception occurs, the original values can be retrieved
 - File must have enough entries for one register modification per cycle for the longest possible instruction
 - Similar to the solution used for the VAX for autoincrement and autodecrement addressing
- Future file
 - This method stores the newer values for registers
 - When all earlier instructions have completed, the main register file is updated from the future file
 - On an exception, the main register file has the precise values for the interrupted state

Handling exceptions: save pipeline state

- Keep enough information for the trap handler to create a precise sequence for the exception
 - Instructions in the pipeline and the corresponding PCs must be saved.
 - After the exception, the software finishes any instructions in the pipeline that precede the latest instruction completed
- State must be saved by hardware with software assist
- Technique used in the SPARC

Instruction ₀	; causes exception	} Pipeline state saved
Instruction ₁	; not completed	
Instruction ₂	; not completed	
...		
Instruction _n	; not completed	
Instruction _{n+1}	; not yet started	

Handling exceptions: delay issue

- Allow instruction to issue only if it is known that all previous instructions will complete without causing an exception
 - Floating point function units must determine if an exception is possible early in the EX stage
 - Necessary to keep the pipeline flowing smoothly (avoid stalls)
 - Pipeline may need to be stalled in order to maintain precise interrupts
 - Solution may cause unnecessary stalls by delaying a sequence of instructions...
- R4000 and Pentium use this solution

ISA and pipelining

- Avoid variable instruction lengths and running times whenever possible
 - Variable length instructions complicate hazard detection and precise exception handling
 - Sometimes it is worth it because of performance advantages such as caching, but this can cause instruction timings to vary
 - Added complexity may be handled by freezing the pipeline
- Avoid sophisticated addressing modes
 - Addressing modes that update registers (post-autoincrement)
 - Complicate exceptions and hazard detection
 - Make it harder to restart instructions
 - Allowing addressing modes with multiple memory accesses also complicates pipelining

ISA and pipelining

- Don't allow self-modifying code
 - Instruction being modified may already be in the pipeline => address being written must constantly be checked
 - Conflict => pipeline must be flushed or the instruction updated!
 - Even if it's not in the pipeline, it could be in the instruction cache..
- Avoid implicitly setting condition codes in instructions
 - Harder to avoid control hazards => impossible to determine if condition codes are set on purpose or as a side effect
 - Implementations that set the CC almost unconditionally make instruction reordering difficult => hard to find instructions that can be scheduled between the condition evaluation and the branch.

Sample pipeline: MIPS R4000

- **IF: first half of instruction fetch**
 - PC selection occurs
 - Cache access is initiated
- **IS: second half of instruction fetch.**
 - Allows cache access to take two cycles
- **RF: decode and register fetch**
 - Hazard checking
 - I-cache hit detection
- **EX: execution**
 - Address calculation
 - ALU Ops
 - Branch target calculation
 - Condition evaluation.
- **DF/DS/TC: data memory**
 - Data fetched from cache in the first two cycles
 - The third cycle involves determine if it was a cache hit
- **WB: write back**
 - Write result for loads and R-R operations

Stalls & delays in the MIPS R4000

- Load delay: two cycles
 - Delay might seem to be three cycles, since the tag isn't checked until the end of the TC cycle
 - However, if TC indicates a miss, the data must be fetched from main memory and the pipeline is backed up to get the real value
- Branch delay: three cycles (including one branch delay slot)
 - Branch is resolved during EX, giving a 3 cycle delay
 - First cycle may be
 - Regular branch delay slot (instruction always executed)
 - Branch-likely slot (instruction cancelled if branch not taken)
 - MIPS uses a predict-not-taken method presumably because it requires the least hardware

Effects of longer pipeline in MIPS R4000

- Disadvantages of a longer pipeline
 - Longer (and possibly more frequent) stalls
 - Additional forwarding hardware
 - More complex hazard detection to find dependencies in the additional stages
- Benefits of longer pipeline
 - Each stage may be shorter
 - Clock cycle can be shorter
 - ⇒ More instructions can be issued in a fixed time
- Do added stalls might eat up this benefit?
⇒ Hopefully, at least some speedup will be left

Performance issues in MIPS R4000

- Ideal CPI for the pipelined CPU is 1
 - Biggest contributor to stalls is branch stalls
 - Load stalls contribute very little (compiler can usually reorganize code to avoid stalling on loads)
 - Load latency is two cycles => job is harder than it might be on processors with a single-cycle latency

Pipelining improves performance!

- Pipelining is one of the main tools designers use to improve performance!
 - Allows the CPU to issue one instruction per cycle even when finishing an instruction takes many cycles
 - Allows faster and faster cycle times by spreading work over many cycles
- Pipelining has been the major factor allowing consumer-level microprocessors to run at 500 MHz or higher
- Next few weeks: more ways to squeeze performance out of the CPU, such as
 - Dynamic optimizations
 - Multiple instructions per cycle