

Topic 2: Basic Pipelining

- **Defining an instruction set:**

1. Data types
2. Addressing Modes
3. Operations
4. Definition of register set

- **DLX: Generic Load/Store Architecture**

1. Data types:

Integer: 8-bit bytes, 16-bit half-words, 32-bit words

Floating point: 32-bit single precision, 64-bit double precision

Fixed-length instructions: 32 bits

2. Addressing Modes

Memory is byte-addressible

Big-endian (bytes number 0, 1, 2, ... in memory; left-most bit represents high-order bits in big-endian machines)

Big-endian machines: SPARC, 680x0 Little-endian: DEC, Intel

32-bit memory addresses

All memory references through loads and stores between memory and either GPRs (general purpose registers) or FPRs (floating point registers)

Can also move between GPRs and FPRs

Only three **data** addressing modes: register, immediate and displacement

TABLE 1.

Mode	Example	Means	Used for
Register	Add R4, R2, R3	$R4 \leftarrow R2 + R3$	Values in register
Immediate or literal	Add R4, R2, #3	$R4 \leftarrow R4 + 3$	for constants; 50% of ALU operations, 85% of compares; most values small
Displacement or based	LW R4, 100(R1)	$R4 \leftarrow M[100 + R1]$	used for accessing local variables; choosing length of displacement field affects instruction length

Register deferred: use displacement mode with displacement = 0 (i.e., register holds a pointer)

Absolute: use displacement mode with base register R0 (tied to 0) (i.e., directly address memory)

- **Branch** addressing modes: PC relative and register indirect
- PC-relative branching
 - Specify an offset relative to the PC, use this to compute target address
 - Why does this work? Target is often near current instruction (examples: loops, switch statements)
 - Use fewer bits to specify
 - Position independence: permits code to run independently of where it is loaded
 - Useful when target of branch is known at compile time (if not, can't use PC-rel)
- Register indirect jumps and branches
 - Jump to a 32-bit address stored in a register
- DLX uses fixed instruction lengths
- This makes for simpler decoding of instructions
- DLX also uses relatively few addressing modes and relatively few combinations
- Can include the addressing mode information as part of the opcode
 - Example: may have ADD and ADDI instructions; first case, expects to see two register operands, second case, expects second operand to be an immediate value, will decode accordingly

3. Operations: 4 classes

(a) Loads and stores

Only one addressing mode: base register plus 16-bit signed offset

Example: LW R1, 30(R2) <<== R1 <- M[30 + R2] (loads 32-bit word)

LW R1, 1000(R0) <<== R1 <- M[1000 + R0] (gives absolute address)

Note: to load an immediate value into a register, use an ADDI rather than a LD operation: use operand R0 and immediate value

(b) ALU operations (ADD, SUB, MUL, AND, OR, Shift, Compares)

Usually have two source register operands and one register destination operand

Example: ADD R2, R1, R3 <<== R2 <- R1 + R3

Also, allow immediate operands

Example: ADD R1, R2, #3 <<== R1 <- R2 + 3

Compares (<, >, <=, >=, ==, !=) on two registers; if true, puts 1 in destination register; if false, puts 0 in destination register

Example: SLT R1, R2, R3 <<== if (R2 < R3) then R1 <- 1 else R1 <- 0

Also have immediate forms of comparisons

(c) Control instructions: branches and jumps

Jumps are unconditional; either (1) specify 26-bit offset added to the PC or (2) specify register containing destination address; may also jump and link (for a procedure call), which places the return address in R31

PC-relative branches add the offset to the incremented PC, or new PC (NPC)

Example: J offset $\lll PC \leftarrow NPC + \text{offset}$

Example: JALR R2 $\lll R31 \leftarrow PC + 4; PC \leftarrow R2$ (save PC+4 for return)

Branches are conditional; test register source for equal to zero or not equal to zero

Example: BNEZ R4, offset $\lll \text{if } (R4 \neq 0) \text{ then } PC \leftarrow NPC + \text{offset}; (16\text{-bit offset})$

(d) Floating point operations (ADD, SUB, Mult, DIV, compares)

Single or double precision; can convert between the two

Example: ADDF F2, F3, F4 $\lll F2 \leftarrow F3 + F4$

4. Definition of register set

32 32-bit general purpose registers

R0 is always 0

R31 holds the return address of a procedure call

32 32-bit floating point registers: 32 single precision or 16 double-precision; double-precision registers accessed by even names (F0, F2,...)

- 32-bit program counter
 - **Register file contains collection of registers: fast temporary storage**
 - Instruction format : 6 bits (up to 64 different opcodes)
1. I-Type (For loads, stores, immediates, conditional branch instructions, jump register)

opcode (6)	RS1 (5)	RD (5)	Immediate (16)
------------	---------	--------	----------------

2. R-Type (For register-register ALU operations)

opcode (6)	RS1 (5)	RS2 (5)	RD (5)	Func (11)
------------	---------	---------	--------	-----------

3. J-Type (For jumps and Jump-and-link)

opcode (6)	Offset added to PC (26)
------------	-------------------------

- **Pipelined Implementation of DLX Architecture**

- ****SLIDE** : Figure 3.4**

- Five stages:

1. Instruction fetch (IF)

$IR \leftarrow M[PC]$

$NPC \leftarrow PC + 4$

Send out PC and fetch instruction from memory (instr. cache) into instruction register

Compute new PC

2. Instruction decode/register fetch (ID)

$A \leftarrow \text{Regs}[IR6..10]$

$B \leftarrow \text{Regs}[IR11..16]$

Fixed-field encoding; since fields describing registers are fixed, can decode them along with instruction and fetch the registers

May read registers don't use; may calculate immediate don't use; doesn't hurt!

Calculate the sign-extended immediate of lower 16 bits of instruction register

3. Execution/effective address step (EX)

- (a) Memory reference: load or store instruction

register A contains base address; instruction contains 16-bit offset

$ALUoutput \leftarrow A + \text{Immediate}$

$SMD \leftarrow B$

Effective address computed with A and offset

If a store operation, data to be stored comes from register B, stored in SMD

- (b) Register-register ALU

$ALUoutput \leftarrow A \text{ op } B$

- (c) Register-immediate ALU

$ALUoutput \leftarrow A \text{ op } \text{Immediate}$

- (d) Branch/jump

$ALUoutput \leftarrow NPC + \text{Immediate}$

$cond \leftarrow (A \text{ op } 0)$

To get branch address, ALU adds PC to sign-extended immediate value to compute the branch or jump target (branch uses 16-bit offset, jump uses 26-bit offset)

For conditional branches, a register set in previous step is compared with zero to see if newly computed address should be inserted into the PC (== or != only)

Since we have load/store architecture, can do address calculation and execution on same step, since no instruction needs to do both

4. Memory access/branch completion (MEM)

ONLY Loads, stores, branches and jumps are active in this stage

(a) Memory

Load: $LMD \leftarrow Mem[ALUoutput]$

Store: $Mem[ALUoutput] \leftarrow SMD$

Given the effective address from ALU, does load or store

(b) Branch and Jump

if (cond) $PC \leftarrow ALUoutput$ else $PC \leftarrow NPC$

If branch condition is true, replace PC with computed address; else, next instr.

For unconditional jumps, condition is always true

5. Write-back stage (WB)

(a) Register-register ALU

$Regs[IR16..20] \leftarrow ALUoutput$

(b) Register-immediate ALU

$Regs[IR11..15] \leftarrow ALUoutput$

(c) Load

$Regs[IR11..15] \leftarrow LMD$

• **Key Pipelining Ideas**

- Hardware technique to overlap execution of multiple instructions
- Divide instruction execution into stages; Put registers between each stage

1. Does pipelining change the latency of a single task?

What is latency? Time from beginning to completion of a task

NO: Pipelining doesn't change latency of executing a single instruction

2. Does improve throughput?

What is throughput? Rate of doing work

Pipelining DOES improve throughput: the rate at which instructions are executed

Why? Because multiple instructions executing simultaneously

3. What is the potential speedup of a pipeline?

Potential speedup is number of pipe stages

What is the maximum possible speedup of the DLX machine?

max speedup = 5, since 5 pipeline stages

We don't achieve this ideal speedup for several reasons:

4. Unbalanced lengths of time in stages reduces speedup (design stages to be same length)
Pipeline rate is limited by the slowest stage
5. Takes time to fill and drain pipeline--reduces speedup
6. Dependencies between instructions may cause pipeline to stall
7. Overhead for storing data in latches between stages

- **Hazards: Dependencies among instructions**

- Keep us from getting full speedup

1. **Structural Hazards**

HW cannot support combination of instructions

e.g., ALU can't compute an effective address and perform a subtract operation at a time

2. **Data Hazards**

Instruction depends on results of prior instruction still in the pipeline

So, have to wait

3. **Control Hazards**

Pipelining of branches and other instructions that change the PC

- **Common solution is to STALL the pipeline until hazard is resolved**
- **Insert one or more BUBBLES in the pipeline (continue plumbing analogy)**

- **STRUCTURAL Hazard**

- **Example: competition for memory**

- ****SLIDE** : Figure 3.6**

- First instruction is a load; three instructions later, try to do an instruction fetch at the same time; can't both access the same memory simultaneously

- **One solution: insert a stall**

- ****SLIDE** : Figure 3.7**

Insert a stall where instr3 would normally have been

Effect: no instruction will finish during cycle 8

- **Another alternative: add hardware to overcome structural hazard**

- In this example, add a second memory; separate instruction and data memories, so that instructions in the pipeline can simultaneously do an instruction fetch and a data access

- Tradeoff: Adding hardware is expensive

- Might allow structural hazards to reduce costs

- Other examples of structural hazards:

not enough memory ports, register file ports, execution units

- As we move to superscalar pipelines with more instructions executing simultaneously, we get more competition for hardware resources

- **DATA Hazards**

- ****SLIDE** : Figure 3.9**

- First instruction does an add, puts result in R1
- Data is not written until the last (write-back) stage
- Next four instructions use this result from R1 as an operand
- For the first three instructions, the data is not ready when they need the operand
- Note: forward arrow is ok -- the XOR instruction has R1 available when it is needed
- One possibility: stall the pipeline for three stages until the result of R1 is available
- Not very desirable
- What is a technique to avoid stalling? Forwarding (we'll look at this in a minute!)

- **Classification of Data Hazards: 3 types**

- Instruction i followed by Instruction j

- 1. Read After Write (RAW)**

As in last example

Instrj tries to read the operand before Instr i writes it

- 2. Write After Read (WAR)**

Instrj tries to write operand before Instr i reads it

Note: this can't happen in the DLX 5-stage pipeline because:

All instructions take 5 stages

Reads are always in stage 2

Writes are always in stage 5

If one instruction comes before another, then the second cannot write before the first one reads it

While WAR doesn't occur in DLX, we will see it in more complicated pipelines that perform out-of-order execution

- 3. Write After Write (WAW)**

Instrj tries to write operand before Instr i writes it

Leaves wrong result--result of Instr i rather than Instrj

Ask: can this happen in the DLX pipeline?

Answer: No, because again, always write in the last stage of the pipeline, so eliminate this kind of hazard

Again, will see WAW hazards in more complex pipeline

- **NOTE: Read After Read (RAR) is not a problem: reads are non-destructive**

- **Many data hazards can be solved by FORWARDING**

- What is it?

- A way of passing results of previous operations to the current instruction
- In many cases, forwarding allows us to avoid pipeline stalls
- We forward values to the inputs of the ALU and sometimes to pipeline registers, for example, the data that is needed for a SW operation
- ****SLIDE** : Figure 3.10**
- ****SLIDE** : Forwarding requires hardware changes: Figure 3.20**
- Additional datapaths from later stages in the pipeline to earlier stages
- More complicated multiplexor on inputs to the ALU
- Note the special case called “forwarding through the register file”: during the WB stage, one instruction can write the value to the register on the first half of a cycle, and another instruction reads the same value on the second half of the cycle
- Forwarding through the register file doesn’t require additional datapaths but increases the complexity of the register file somewhat
- **Forwarding does not solve all data hazards (Figure 3.12)**
- The one case that still requires a stall is a LW instruction followed immediately by an instruction that reads the value being loaded
- First instruction loads R1; data value not ready until after memory access in stage 4
- Next instruction uses R1; needs value at start of stage 3
- ****SLIDE** : Must stall for one cycle (Figure 3.13)**
- **Can’t avoid the stall with hardware**
- **But, a smart compiler can help**
- **Rearrange the code: “scheduling”**
- Try producing fast code for: a=b+c; d=e-f; assume a, b, c, d, e, f all in memory
- Slow code:
 - LW Rb, b
 - LW Rc, c
 - ADD Ra, Rb, Rc <<<----- Tries to use Rc at start of cycle 4; not ready until end of C4
 - SW a, Ra <<< ----- Dependency but not hazard, since hardware allows forwarding
 - LW Re, e
 - LW Rf, f
 - SUB Rd, Re, Rf <<<----- Tries to use Rf a cycle before it is available
 - SW d, Rd
- **Question: where are the hazards in this code?**

- **How to SCHEDULE or rearrange code to make it fast/avoid all stalls?**
- Fast code:
 - LW Rb, b
 - LW Rc, c
 - LW Re, e
 - ADD Ra, Rb, Rc
 - LW Rf, f
 - SW a, Ra
 - SUB Rd, Re, Rf
 - SW d, Rd
- **How successful are compilers at inserting instructions to avoid hazards?**
- ****SLIDE** : (Figure 3.16) (Don't show)**
- Percentages of loads that cause stalls after compiler scheduling for ten SPEC benchmarks; FP average 13%, integer average 25%
- **Control hazards: Branches**
- Remember: calculate zero compare and effective address in third stage
 - Don't know about branch until end of stage 2
 - Can't affect PC until 4th stage (MUX)
- Potentially, have to stall 3 cycles to resolve branch
- What is the impact of this?
- **Figure 3.21**

Branch Instruction:	IF	ID	EX	MEM	WB
Branch Successor:		IF	stall	stall	IF
		^		^	
		__ actually write to PC after the MUX			
		Fetched but ignored			
- This long branch penalty has a big impact on performance!
- **The CPU Performance Equation:**
- **Want to calculate the execution time of a program given a particular pipeline**
- **What information do we have/need?**
- Computers divide execution time into clock cycles
- We know speed of clock, so know seconds per cycle
- We know program being executed, so know # of instructions

- **We can calculate the average number of cycles per instruction**
- $\text{CPUTime} = \text{Seconds/Program} = \text{Instructions/prog} * \text{Cycles/Instr} * \text{Seconds/cycle}$
 $= \text{instruction count} * \text{CPI} * \text{cycle time} = \text{IC} * \text{CPI} * \text{CT}$
- Note: Clock Rate = 1 / cycle time
- Only one safe measure of computer performance: time to execute real programs

- How to calculate CPI (Cycles per Instruction)?
- $\text{CPI} = \text{Sum}_{(i=1 \text{ to } n)} (\text{CPI}_i * F_i)$
 where $F_i = \text{frequency of instruction } i = \text{Num Instr } i / \text{Total Instr Count}$
- Note: CPI depends on instruction mix, so will change with workload (Pick typical)
- Average over all instructions in program

- **AN EXAMPLE:**

OP	Frequency	CPIop
ALU	.43	1
load	.21	2
store	.12	2
branches	.24	2

- $\text{CPI} = .43 (1) + .21 (2) + .12 (2) + .24 (2) = 1.57$

- **In an ideal DLX pipeline (with no hazards), what is the CPI?**
- **CPI = 1, since on average one instruction enters and one leaves the pipeline every**
- **Note that instruction latency is not 1: latency = 5 cycles for DLX pipeline**
- **CPI is an average measure, a reflection of throughput of instructions**

- **Problem: branch delays increase the cpi**

Ideal CPI = 1

Assume 30% of instructions are branches

If branch causes the pipeline to stall for 3 cycles, then branches take total of 4 cycles

Then $\text{CPI}_{\text{branch}} = 4$ and $\text{CPI}_{\text{other}} = 1$

New $\text{CPI} = (0.7)(1) + (0.3)(4) = 1.9$

$\text{CPUtime} = \text{CPI} * \text{IC} * \text{CT}$ -- Execution time of programs almost doubles

- **Want to lessen the branch penalty: Three-part solution**

1. **Determine whether we take the branch or not SOONER (branch condition)**
2. **Computer the branch target address sooner**
3. **Change PC sooner**

- In DLX: Take care of branch in ID/RF stage
- Move zero test to ID/RF (calculate branch condition)
also need a second adder to calculate branch target in ID/RF (calculate branch target)
also move MUX to change the PC there (change PC)
- Result: **1 cycle branch penalty** instead of 3
- CAUTION: Potentially lengthen the clock cycle because of additional gate delays
- (If going to branch, wasted one instruction fetch)
- **** SLIDE ** 3.22* Shows new hardware**
- Still have one cycle branch delay in DLX: need to try to improve this
- Large, deeply pipelined machines may have branch penalties of 6 or 7 cycles

- **Before looking at how to DEAL with branches, first understand branch behavior**

- **Understanding Branch Behavior**

- **To do this, we use the SPEC benchmarks**

- **ASIDE: What are benchmarks?**

- **Collections of programs that are run on machines to test their performance**

- Varying degrees of “realness”:

real programs

kernels: small but important parts of real programs

synthetic benchmarks: not real programs; intended to mimic the instruction mix of a large set of programs

benchmark suites: collections of benchmarks designed to measure performance for a variety of applications

- **SPEC benchmarks used in this book (performance evaluation consortium?)**

- **See Figure 1.9 for details on Spec92 benchmarks used in this book**

- **Branch statistics for SPEC Programs shown in Figures 3.24**

TABLE 2. Figure 3.24: Frequency of Instructions that Change the PC

	Forward conditional	Backward conditional	Unconditional
Integer	13%	3%	4%
Floating Point	7%	2%	1%

- So, branches are about 20% of integer instructions, 10% of floating point benchmarks

- **Are conditional branches usually taken or not? What do you think?**

- **Taken vs. untaken: Figure 3.25**
- 67% of conditional branches are taken (62% integer, 70% FP)
- **Overall:**
 - **60% of forward branches taken**
 - **85% of backward branches taken (loops!)**
- **Based on this understanding of branch behavior...**
- **We would like to reduce frequency of stalls due to branches**

- **What are ways to deal with branches?**

1. Freeze/stall pipeline until branch outcome is resolved

As soon as branch is decoded, stall the successor instruction until the branch is resolved
 Shown in Figure 3.21 for original pipeline

For the new pipeline where branches are resolved in the ID phase:

branch	IF	ID	EX	
successor		IF	XXX		(abort this instruction if branch is taken)
branch target			IF		

Simplest scheme but doesn't reduce number of stalls

2. Predict branch not taken

This is a prediction made at compile time

Fairly simple to implement

PC+4 already calculated in the pipeline; continue execution

If prediction is wrong, back out: turn the successor instruction into a NOOP

Be careful not to change state of machine until branch outcome is known

Causes one cycle stall if branch taken, no stall if not taken

- **Figure 3.26**

IF	ID	EX		
	IF	IF	ID (fetch next instruction, throw away if prediction wrong)

In DLX, if wrong, reset fetched instruction to NO-OP and fetch next instruction

3. Predict that branch is taken

Prediction made by compiler

Fetch from target as soon as it is known

Should be a good prediction: statistics showed that most branches are taken

But this technique doesn't help in DLX. Why not??

Answer: find out whether branch is taken and target address on same cycle

Predict taken is a useful technique when target is known before condition

- **Example, p. 174:**
- 3 stages before branch target is known
- IF ID EA BC
- one more cycle before branch condition is evaluated
- Branch penalties for schemes:

TABLE 3. Penalties for prediction schemes (Figure 3.34)

Branch scheme	Penalty Unconditional	Penalty untaken	Penalty taken
Flush pipeline	2.0	3	3
Predict taken	2.0	3	2
Predict untaken	2.0	0	3

- IF ID EA BC
- IF ID EA BC ---- predict untaken (correct)
- IF ID EA BC
- IF ID EA BC ---- unconditional branch, predict taken (correct)
- IF ID EA BC ---- flush pipeline conditional, incorrect prediction

4. Delayed branch

Again, depend on the compiler

We make the length of the branch delay visible to the compiler

In case of DLX, compiler knows that there is a branch delay penalty of one cycle

Compiler tries to schedule or re-arrange the code to put useful instructions in the branch delay slots

Note: the instruction in the branch delay slot will ALWAYS be executed, whether or not the branch is taken

No more killing instructions after branch is decoded

Question: can compiler always find something useful to put into that slot?

No, sometimes must fill slot with NOOP

Similar to scheduling to reduce load hazards

- **Where do instructions come from for filling the slots?**
- Figure 3.28 in the book
- 1. Before the branch instruction: ALWAYS helps
- 2. From the target of the branch: Helps when branch is taken
- 3. From fall-through: Helps when branch is not taken

- **Instructions put in the branch delay slot must be SAFE**
- **If outcome is different from prediction, then we may waste work but we may not introduce errors**
- **Cancelling branch**
 - Used along with delayed branch
 - Add a prediction of the branch outcome to the branch instruction
 - When branch prediction is correct, we complete execute branch delay slot instruction
 - Otherwise, turn instruction in branch delay slot to NOOP
 - Allows more flexibility in putting instructions into the branch delay slot that otherwise might not be “safe” (i.e., would introduce errors if they were allowed to complete after an incorrect prediction)
- **Figure 3.31: use of delay slots and cancelling branches**
 - Use traditional delayed branch to put “safe” instructions into branch delay slots when possible
 - If not, use cancelling branches
 - Overall, even with two techniques, still 20% of normal branch delays filled with NOPs
 - Because candidates for the delay slot are unknown (e.g., large case statement) or subsequent instr. are branches
 - Of the other 80%: 41% use canceling branches, 59% use delay slots
 - Of the cancelling branches, 35% are cancelled (65% do useful work)
- **Figure 3.32; overall effectiveness of filling branch slots**
 - Overall, 70% of branch delay slots usefully filled
 - (Combination of delay slots and cancelling branches)
 - So, branch penalty is 0.3 cycles per conditional branch
- **Disadvantage of delayed branch: visible part of the architecture (so compiler can use it) but likely to change)**
- **Continued: Example on P. 174**
 - Find the effective addition to the CPI from branches using SPEC benchmarks
 - SPEC integer: 4% unconditional, 10% conditional taken, 6% conditional untaken

- ***Note and error in book, untaken and taken percentages are backwards***

TABLE 4. Addition to the CPI

Scheme	Unconditional branches	Untaken conditional	Taken conditional	All Branches
Freq. of event	4%	10%	6%	20%
Stall pipeline	0.08	0.30	0.18	0.56
Predict taken	0.08	0.30	0.12	0.50
Predict untaken	0.080	0.00	0.18	0.26

- Contribute to cycles per instruction: compared to ideal machine, $cputime = old + .2 * extra$

- **How does the compiler make predictions about whether a branch is taken or not?**

1. Static prediction

- **Most branches (forward and back) are taken, so predict taken**

Average misprediction rate 34%

But wide variance

- **Predict backward branches (loops) likely to be taken**

Predict forward branches not taken

Not very good; misprediction rate 30-40%

Difficult to make a better guess just based on program structure

2. Profile-based prediction

Predict future based on past

Figure 3.36

Much better predictor: branches often either mostly taken or mostly untaken

Misprediction rates on SPEC: 9% on FP, 15% on integer

problem: not always convenient to run everything twice to make predictions

- **Brief overview of hardware implementation:**

1. Detecting hazards

2. Inserting stalls

3. How does machine detect that instructions inserted by compiler must be discarded

- **Detecting Hazards**

- For DLX pipeline, all data hazards can be checked during ID phase

- Also determine necessary forwarding in ID, set appropriate control signals

- Detecting early reduces hardware complexity:

Don't allow conflicting instructions beyond this stage,

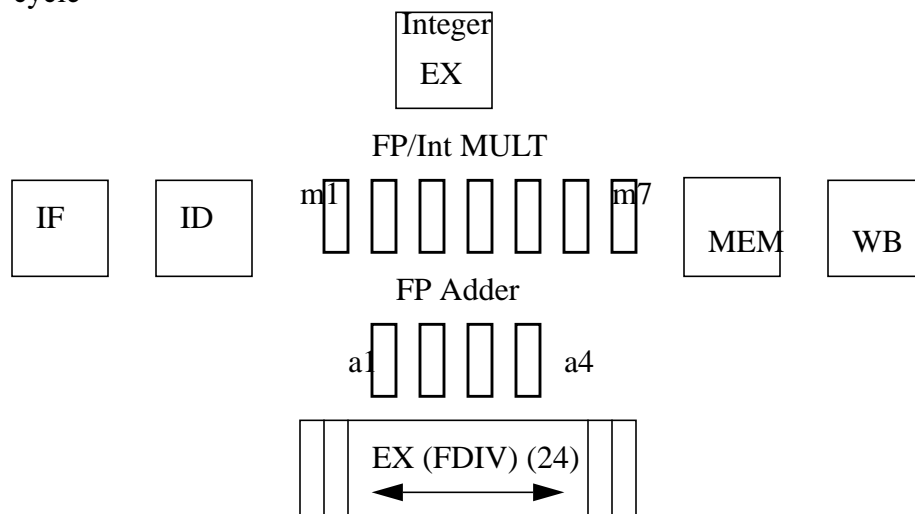
So never have to worry about instruction that changes state and is then aborted

- Alternative: detect hazard or forwarding at start of cycle that uses the operand (MEM or EX for DLX)
- **Hazard detection looks at register fields in instructions, decides (Fig. 4.18, 4.19):**
- No dependence
- Dependence requiring a stall
- Dependence overcome by forwarding
- Dependence but no action needed because hardware will execute in correct order
- **Stalling the pipeline**
- Once hazard is detected, insert a stall
Prevent instructions in IF, ID from advancing
- All control info is contained in pipeline registers
- Change control part of pipeline to all zeros (No-OP)
- Recirculate contents of ID/EX and IF/ID registers to hold state of two instructions that are stalled
- **Implementing forwarding**
- Similar logic to detect dependencies, then set appropriate control paths
- **Implementation of compiler filling branch delay slots**
- Compiler will insert instructions into delay slot
- Pipeline will fill up with predicted instructions;
then check PC against branch outcome to determine whether prediction was correct
- Example: predict not take, will continue to fetch normal instruction sequence
If wrong (branch taken):
Turn fetched instruction into No-Op by clearing IF/ID registers
Restart fetch with target address
- **Floating point**
- Up to now, our execution stage has been 1 cycle long
- Problem for floating point instructions: take a long time to execute
- Potentially very long EX phase
- Options: (1) increase clock cycle time (slow down all instructions)
(2) use multiple cycle execution for floating point instructions
- Generally implement option (2)
- Think of this as repeating the EX stage multiple times

- Problem: a single instruction repeating EX stage will hold up entire pipeline
- Solution: Multiple execution units that allow several instructions to EX simultaneously
- ****SLIDE** : Figure 3.42: Multiple pipelines, repeated execution stages**

- **Multiple Execution Units**

- We implement this as multiple execution units with different components or stages
- Example: break the floating point ADD into four stages
- Takes four cycles to compute the floating point ADD result; 8 total cycles to compute
- Execution unit includes four pipeline stages with a pipeline register between each stage
- Call this adder “fully pipelined”, meaning a new ADD instruction can enter the pipeline at every cycle



- **Floating point/integer multiplier is also fully pipelined with 7 stages**

Takes 7 + 4 cycles to get through the pipeline
 New multiply instruction can enter the pipeline every cycle

- **Division unit not pipelined**

Takes 24 cycles to execute; only one divide instruction can be active at a time
 Highly dependent operations, difficult to break into unique stages

- If another divide comes along, **will not have the hardware resources to execute**
 What kind of hazard or dependency is this? (structural)

- **Introduce several new hazards**

1. **Structural hazards**

- Divide unit not fully pipelined; subsequent divide operations have to wait

- Competition for register accesses (multiporting expensive)
- Competition for data memory
 - May have several instr completing on same cycle
 - Solution: stall all but one
 - Heuristic (simple): allow longest-running instruction to proceed (likely bottleneck)

2. RAW Hazard:

Short instruction may need result of previous long-running instruction

Solution: detect hazard in ID and stall

May stall for many cycles!

3. WAW Hazard:

WAW hazards do occur when instructions complete out-of-order

Example: ADD instruction followed two cycles later by a load instruction

Add F2, F4, F6: IF ID A1 A2 A3 A4 MEM WB

--- IF ID EX M WB

LW F2, 0(R2): IF ID EX M WB ****WAW hazard****

Solution: detect hazard in ID and stall, delay “issue” of instruction to execution unit

- **DO NOT introduce WAR hazard (still read early, write late)**

• EXCEPTIONS: Handling Exceptions in Pipelines

- Exceptions: stop normal stream of execution and take care of something more urgent
- Hard in pipelines because multiple instructions are in progress
- In many cases, need to restart after dealing with exception or interrupt

• ****SLIDE****: Table on P. 179: Types of Exceptions

I/O device request (restart)

Breakpoint (restart)

Arithmetic underflow or overflow (terminate)

Page fault (restart)

Memory protection violation (resume)

Power failure (terminate)

Hardware malfunction (terminate)

- **Exceptions in the DLX Integer pipeline**
- ****SLIDE** : Figure 3.41**

TABLE 5.

Stage	Problem interrupts
IF	Page fault on IF Memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic interrupt (overflow? divide by 0?)
MEM	Page fault on DF Memory protection violation

- DLX: 5-stage pipeline: 5 instructions executing
 - How to stop pipeline? How to restart?
1. Halt pipeline and save state
 - Force a trap instruction into pipeline on next IF
 - Disable writes for faulting instruction and all following instructions to save state
 - Turn faulting instruction and following instructions into NOOPS
 - Store PC of faulting instruction
 - If delayed branch is used, save PC of instructions in delayed slot (compiler may have inserted instructions from elsewhere after the branch)
 - Total: save number of delay slots plus one for PC
 2. Service Interrupt
 - Trap triggers interrupt processing routine in operating system
 3. Restore state and restart
 - Re-load PC of faulting instruction and instructions in branch delay slots
- **Precise vs. imprecise interrupts**
 - Precise interrupts: stall pipeline so that
 - all instructions before the faulting instruction complete
 - all instructions after faulting instruction restart from scratch
 - faulting instruction changes no state
 - Difficult in pipelined machines, especially with long-running instructions

- Impact of precise interrupts: slow down execution
Limits the amount of instruction overlap and out-of-order execution
- Solution: two modes of operation
 1. Imprecise interrupts: better performance but may introduce errors
 2. Precise interrupts: slower but better precision on math calculations
- **Imprecise interrupts**
 - Handle interrupts as they occur
 - Not necessarily in same order as instructions
- **Precise Handling of multiple interrupts from different instructions (DLX integer pipeline)**
 - Example: LW gets a data page fault (cycle 4--MEM) followed by
ADD gets an instruction page fault (cycle 1--IF)
 - If serviced as they occur, ADD instruction page fault will be serviced first
 - In some cases, may change results--change precision of calculations
 - To maintain precise interrupts, we need to handle all interrupts from the LW instruction before we handle any interrupts from the ADD instruction
 - Associate an interrupt status vector with each instruction (part of pipeline register)
 - Post interrupts for a particular instruction to its status vector as they occur
 - At start of WB stage, when instruction execution is almost complete, handle interrupts for the instruction in the order in which they occurred
 - This forces a precise ordering on the interrupts
- **Interrupts more complicated with long-running instructions**
- **Out-of-order completion**
 - Example: DIVF F2, F10, F6 followed by ADDF F10, F10, F8
 - Interrupt can occur after ADDF completes (destroying operand F10) but before DIVF completes
 - If instr. finish out-of-order, we may destroy the operands of a long-running instruction
 - How to restart?
 - Example: floating point divide operation:
may take many cycles; may finish out of order;
in the meantime, a shorter ADD instruction may destroy operands of DIV
If DIV instruction later interrupts, how can it be restarted?
How can the state changes of the ADD instruction be undone?

- Can't simply re-start instructions after the fault
- Possible solutions:
 1. Imprecise interrupts -- not acceptable today
 2. Hardware solution: expensive: buffer results, write in correct order
 - a) History file keeps original values of registers
 - b) Future file keeps newer value of register for later restart
 3. Allow partially imprecise interrupts,
 - keep enough state info so that trap-handling routines can create a precise sequence:
 - keep state to finish unfinished instructions,
 - example: what instructions were in pipeline, what were PCs
 - After handling exception: skip over finished instructions, restart others
 - May need to limit overlap of FP instructions to keep it tractable
 - somewhat simplified in DLX because only have to worry about unfinished FP ops;
 - if one integer operation completes, all previous integer operations will have completed
 4. Limit instruction use:
 - issue new instructions only if can detect that preceding instructions will not cause interrupts;
 - detect early and stall