Processor Architecture: In-Order Pipelines

Cosmin E. Oancea cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

September 2024 PMPH Lecture Notes

Course Organization

W	HARDWARE		SOFTWARE	LAB/CUDA
1	Trends Vector Machine	 ⟵	List HOM (Map-Reduce)	Intro & Simple Map Programming
2	In Order Processor	⟶ ⟵	VLIW Instr Scheduling	Scan & Reduce
3	Cache Coherence		Reasoning About Parallelism	Sparse Vect Matrix Mult
4	Interconnection Networks		Case Studies & Optimizations	Transpose & Matrix Matrix Mult
5	Memory Consistency		Optimising Locality	Sorting & Profiling & Mem Optimizations
6	OoO, Spec Processor		Thread-Level Speculation	Project Work

Three narative threads: the path to complex & good design:

- Design Space tradeoffs, constraints, common case, trends.
- Reasoning: from simple to complex, Applying Concepts.

Acknowledgments

This lecture presents selected Topics from Chapter 3 of the "Parallel Computer Organization and Design" book, by Michel Dubois, Murali Annavaram and Per Stenstrom.

Overview

- Processor (& ISA): “the brain”, drive the architectural design.
- Will recall MIPS and exceptions, which, albeit implemented in software, impose architectural constraints.
- Starting Point: 5-stage pipeline, executing instructions in program order. At this level we look at how to best exploit ILP.
- Mechanisms to handle hazards: forwarding, stalling, flushing.
- *Extensions* of the classic 5-stage static pipeline:
  - superpipelined: deeper pipelines, clocked faster, supporting more complex instructions, e.g., floating point,
  - superscalar: fetch and execute multiple instrs in each cycle,
- Core Multi-Threading: a simple solution to latency hiding,
- Vector Machines.

Overview Continuation

- Dynamically Scheduled Out-of-Order (OoO) Processors are NOT covered until the end of the course!
  - daunting task because instructions are executed out of order, but data dependencies, conditional branch outcome, and exceptions must be correctly processed (as if in program order).
  - requires complex hardware support for speculative execution.

- Complexity of OoO Processors prevents wide-superscalar pipelines $\Rightarrow$ next lectures on software track studies compiler optimizations to keep hardware simple & clock rates high:
  - VLIW: simple hardware allowing a large number of instrs to be processed every cycle: data dependencies, speculative execution and exceptions are handled statically (i.e., in software).
  - Vector Processors: compiler restructures code into vector instructions, in which the same instruction is applied on many scalar operands. Can be deeply pipelined at high-clock rates.

ISA: RISC vs CISC

Reduced Instruction Set Architecture (RISC):

ISA: RISC vs CISC

Reduced Instruction Set Architecture (RISC):

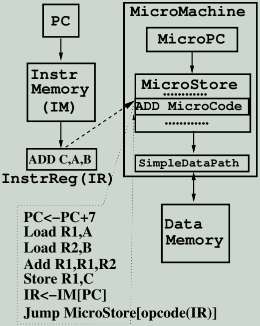
- start with the minimum # of primitive instrs & add only if justified by performance gains.
- Constant instr size, regular formats $\Rightarrow$ simplified decoding, promotes pipelining & short cycle time.

Complex Instruction Set Architecture (CISC):

- Premises: compact instr stream $\Rightarrow$ effective time/space instr-memory utilization $\Rightarrow$ variable instr length and formats.
- Misguided tendency to provide complex ISA to help assembly programmers or to compile high-level languages.
- DEC Vax-11 ISA successful in the 70-80s.
  Intel iAPX432 1980s zenith of complexity: >200 opcodes, 6-300 bits instr length with Huffman coding, in the context of ADA.

ISA: RISC vs CISC (Continuation)

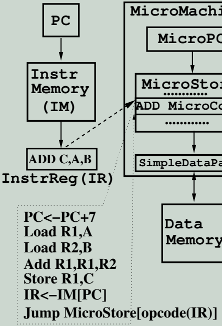
Execution of a CISC instruction



Complex instr fetched from `InstrMemory` & decoded:

ISA: RISC vs CISC (Continuation)

Execution of a CISC instruction



Complex instr fetched from `InstrMemory` & decoded:

- opcode and operand types determines the entry in `Microstore` where its sequence of μinstrs reside,
- μinstructions are similar to the one of MIPS,
- the last μinstruction execution directs the `InstrMemory` to fetch the next complex instr.
- Inefficient execution because μprogram exhibit little ILP & extra cycles are required for the interpretation of ANY instr (simple or complex).
- RISC: removes the interpretation overhead by compiling directly to μcode, which is exposed to compiler optimizations.

- RISC & CISC refers NOT to hwd simplicity (cost of implementing CISC is marginal). Today architectures are CISC due to legacy concerns, but morally RISC:
- CISC instr translated to a sequence of RISC, executed by a RISC-ISA mechanism.
- CISC: IBM System370..., IntelX86 (Pentium4,AMD Turion), Motorola 68000,
- RISC: SunSPARC(SPARCT2), PowerPC(601 IBM), Alpha(DEC), IA64(Itanium2).

MIPS Instruction Set

Instructions in 32-bit format aligned in memory. Addresses assumed aligned (why?). Each instr has one opcode and up to three operands:

Arithmetic/Logic: use integer registers `R0...R31`. Unsigned (`ADDU`,`SUBU`,`MULTU`) and logical (`OR`,`AND`,`NOR`,`NAND`) instructions do not raise exceptions. Signed instr (`ADD`,`SUB`,`MULT`) raise under/overflow exceptions.
`ADD R1,R2,R3` $\equiv$ `R1`$\leftarrow$`R2+R3` and `ADDI R1,R2,#8` $\equiv$ `R1`$\leftarrow$`R2+8`.
`SLT R1,R2,R3` $\equiv$ `R1` $\leftarrow$ `if R2 < R3 then 1 else 0`.

Floating Point: Numbers in sign-exponent-mantisa widens the range of representable numbers. Use float registers `F0...F31`. Can be executed as procedures/macros, but much faster in hardware. No immediate field! Single (`ADD.S`,`SUB.S`,`MUL.S`) and double (`ADD.D...`) precision instrs.
A double register is hold in 2 consecutive registers and is even numbered:
`MUL.S F1,F2,F3` $\equiv$ `F1`$\leftarrow$`F2*F3` and `ADD.D F0,F2,F4` $\equiv$ `F0`$\leftarrow$`F2+F4`.

Memory Access: move operands from/to memory: 1 value reg & 1 address reg & 1 displ. `LB`,`LH`,`LW`,`LD`,`L.S`,`L.D` loads a byte, half/single/double integer word, and a single/double float. Similar for store `SB`,`SH`,`SW`,`SD`,`S.S`,`S.D`:
`LW R1,8(R2)` $\equiv$ `R1`$\leftarrow$`MEM[R2+8]` and `SW R1,8(R2)` $\equiv$ `MEM[R2+8]`$\leftarrow$`R1`.

Branch/Jump: `BNE R1,R2,loop` branches when `R1`$\neq$`R2`; branches use PC-relative targets
`SLT R1,R2,R3; BEZ R1,loop` $\equiv$ `if R2` $\geq$ `R3 then goto loop`.
Jump to an immediate-field (`J target`) or to a register target (`JR R1`) sets PC to target's value. Jump-And-Link (`JAL target`), used for procedure calls, first saves the return address in `R31` then jumps.

Instruction Formats

Most instructions may be executed in one cycle; complex ones require more cycles, e.g, integer/float multiplication/division.

Operand addresses must be aligned (address is a multiple of its size) ⇒ simplifies mem interface (operand cannot span two cache lines).

Three Instruction Formats

31 26	25 21	20 16	15 11	10 0	
opcode	Rs	Rt	Rd		ADD Rd,Rt,Rs

31 26	25 21	20 16	15 0	

opcode	Rs	Rt	displacem/immediate/offset	LW Rt,displ(Rs)
				SW Rt,displ(Rs)
				ADDI Rt,Rs,immed
				BEQ Rt,Rs,offset

31 26	25 0	
opcode	target	J target JAL target

Instruction Mixes

Static Mixes: the distribution of instruction types in a program, e.g., used to determine the size of instruction caches.

Dynamic Mix: refers to the number of executed instructions, e.g., instructions in a loop are repeatedly executed. Used as a primary guideline in processor design.

Table Below shows a rough estimate of the dynamic mix of integer benchmarks:

OpCode Class	Fraction	CPI
Load	25%	high
Store	12%	low
ALU	40%	low
Branches	20%	high
Jump	2%	low
JAL	1%	low

Instruction Mixes

Static Mixes: the distribution of instruction types in a program, e.g., used to determine the size of instruction caches.

Dynamic Mix: refers to the number of executed instructions, e.g., instructions in a loop are repeatedly executed. Used as a primary guideline in processor design.

Table Below shows a rough estimate of the dynamic mix of integer benchmarks:

OpCode Class	Fraction	CPI
Load	25%	high
Store	12%	low
ALU	40%	low
Branches	20%	high
Jump	2%	low
JAL	1%	low

Loads high CPI is due to the memory/bandwidth wall.

Branches high CPI is because they break the predictability of the segment of instructions fetched in the pipeline.

Processor Design must address these and other issues.

Exceptions

- Rare events triggered by hardware & direct the processor to execute a handler.
- Similar with branches (share same hardware), but the difference is that the exception handler is NOT explicitly scheduled in code.
- Exception handling is part of ISA specification:
  - Imprecise: e.g., hwd failure, mem-access violations, are NOT synchronized with an instruction because process is terminated after handler.
  - Precise: e.g., under/overflow, page faults @ instruction `i`. Instrs preceding `i` must finish & instrs `i` and following aborted. Execution resumed from instr `i`.
- Precise exceptions place constraints on hardware & compiler:
  - exceptions are rare and unpredictable $\Rightarrow$
  - no need to speed up the handling BUT build the mechanism & minimize its common-case (conservative) overhead.

Exceptions, Interrupts, Traps

- I/O device interrupts, e.g., DMA
- Operating Systems Calls (TRAP instruction similar to JAL).
- Instruction Tracing & Breakpoints, e.g., CPU traps every instr, valuable info for arch design.
- Integer/Floating Point Exceptions (process resumed or not)
- Page Faults (CPU trapped to the kernel).
- Misaligned Memory Accesses
- Memory protection violation, e.g., out-of-bounds, access rights
- Undefined instruction, e.g., used to extend ISA in software,
- Hardware & power failure.

1. Instruction Set Architectures (ISA)
   - RISC vs CISC
   - MIPS Instruction Format & Mixes
   - Exceptions
2. Static 5-Stage Pipeline
   - Naive Architecture
   - Resolving Data Hazards for the 5-Stage Pipeline
   - Resolving Control Hazards for the 5-Stage Pipeline
   - Handling Precise Exceptions in the 5-Stage Pipeline
3. Out-Of-Order Instruction Completion
   - Data, Control, Structural Hazards & Exceptions
   - SuperPipelined and SuperScalar CPUs
   - Branch Prediction
   - Strength & Weaknesses of Static Pipelining
4. Core Multi-Threading
   - The Barrel Processor
   - Interleaved Core Multi-Threading
5. Vector Machines

Classic RISC 5-Stage Static Pipeline

- Pipeline improves instruction throughput. Think assembly line in which every worker/robot works on and is specialized for one stage only. Have you ever moved to another apartment?

- Pipeline stages need to be very similar. Instruction execution good candidate

Classic RISC 5-Stage Static Pipeline

- Pipeline improves instruction throughput. Think assembly line in which every worker/robot works on and is specialized for one stage only. Have you ever moved to another apartment?
- Pipeline stages need to be very similar. Instruction execution good candidate because goes through same stages: fetch, decode, execute, write results.
- RISC well suited for pipelining because efficiency requires minimal differences in format & execution of various instruction types.
- Main Problem

Classic RISC 5-Stage Static Pipeline

- Pipeline improves instruction throughput. Think assembly line in which every worker/robot works on and is specialized for one stage only. Have you ever moved to another apartment?
- Pipeline stages need to be very similar. Instruction execution good candidate because goes through same stages: fetch, decode, execute, write results.
- RISC well suited for pipelining because efficiency requires minimal differences in format & execution of various instruction types.
- Main Problemis to handle efficiently data and control dependencies between instructions. Techniques that alleviate such negative effects are constrained by exception handling.

The 5 Stages of the Classic RISC Pipeline

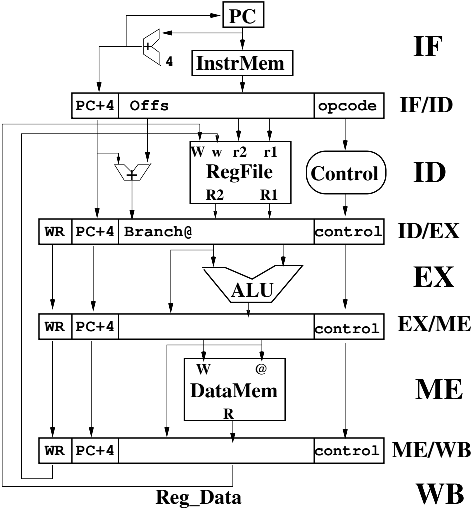
Instruction	I-Fetch IF	I-Decode ID	Execute EX	Memory ME	WriteBack WB
LW R1,20(R2)	fetch; PC+=4	decode; fetch R2	compute addr (R2)+20	read mem	write in R1
SW R1,20(R2)	fetch; PC+=4	decode; fetch R1 and R2	compute addr (R2)+20	write mem	-
ADD R1,R2,R3	fetch; PC+=4	decode; fetch R2 and R3	compute (R2+R3)	-	write in R1
ADDI R1,R2,imm	fetch; PC+=4	decode; fetch R2	compute (R2+imm)	-	write in R1
BEQ R1,R2,offs	fetch; PC+=4	decode; fetch R1 and R2 and in || (PC+offs) as the target address	compute (R1-R2); take branch if 0	-	-
J target	fetch; PC+=4	decode; take branch	-	-	-

- Pipeline improves instr throughput. Think assembly line in which every worker (robot) works on and is specialized for one stage only. Switched appartments?

- IF is the same. ID: 2 regs (always) fetched + target address computed for branch.
- EX computes address, or values, or compares registers and take or not a branch.
- ME is active for loads and stores (for others NOOP). WB updates output reg.
- floating point and MUL/DIV do not fit yet ⇒ subroutines.

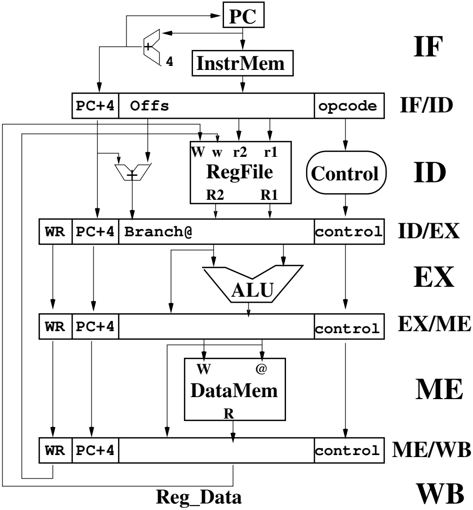
- Instructions move through pipeline in program order!
- CPU is frozen on a cache miss! Clock is stopped & resumed after miss is serviced.

Classic RISC Pipeline Architecture



- Data-Path Resources: instr memory (cache), register file (2 read & one write ports), ALU & data mem (cache)
- Pipeline Registers, clocked every cycle, separate any 2 consec stages (ID/EX)
- Clean Pipeline Design:

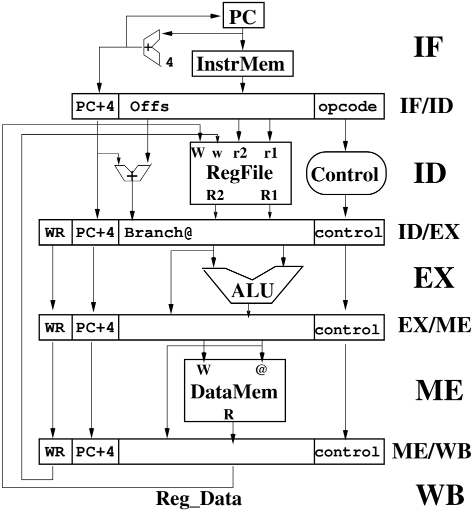
Classic RISC Pipeline Architecture



- Data-Path Resources: instr memory (cache), register file (2 read & one write ports), ALU & data mem (cache)
- Pipeline Registers, clocked every cycle, separate any 2 consec stages (`ID/EX`)
- Clean Pipeline Design: instr is recoded as it moves through pipeline, and carries the whole info needed to complete execution.
- IF: every clock PC+4 and in parallel current instr is fetched from cache. At the trailing edge of the clock PC is updated and instr is stored in `IF/ID`,
- ID: opcode decoded to *control signals*, which will be connected to and (de)activate various units of `EX`,`ME`,`WB`. Two input registers are always fetched, PC+4 carried in case of exception.

This design assumes independent load, store, and ALU instructions, i.e., instructions do not share registers or memory locations.

Classic RISC Pipeline Architecture



- EX: control signals of EX are applied and stripped. One input reg connected to ALU. 2nd ALU input is either a register or, not shown, the 16 least significant digits (immed/displ). For load/stores ALU computes addresses. The value to store bypasses ALU.
- ME: control signals of ME are applied and stripped. For load/stores the address is the ALU output, and feeds into the address bus of the memory. The value to store connected to the input data bus. Values are stored/loaded from mem at trailing edge. ALU instrs bypass ME stage.
- WB: remaining control signals applied. The mem value (for load) or from ALU is stored to output register at the trailing edge of the clock. WR field used to index the register file.

This design assumes independent load, store, and ALU instructions, i.e., instructions do not share registers or memory locations.

Data Hazards for the 5-Stage Pipeline

Data Hazards: Property of Software

```
RAW (True Dependency)    WAR (Anti Dependency)    WAW (Output dependency)
S1    X  = ..            S1    .. = X             S1    X = ...
S2    .. = X             S2    X  = ..            S2    X = ...
// S2; S1 gives a different result than S1; S2 => Hazard
```

What hazards can occur in the 5-Stage Pipeline?

- Dependencies on memory operands

Data Hazards for the 5-Stage Pipeline

Data Hazards: Property of Software

```
RAW (True Dependency)     WAR (Anti Dependency)     WAW (Output dependency)
S1    X  = ..             S1    .. = X              S1    X = ...
S2    .. = X              S2    X  = ..             S2    X = ...
// S2; S1 gives a different result than S1; S2 => Hazard
```

What hazards can occur in the 5-Stage Pipeline?

- Dependencies on memory operands do not cause hazards because loads/stores are executed in ME stage in program order.
- WAW hazards

Data Hazards for the 5-Stage Pipeline

Data Hazards: Property of Software

```
RAW (True Dependency)      WAR (Anti Dependency)      WAW (Output dependency)
S1    X  = ..              S1     .. = X              S1     X = ...
S2    .. = X               S2     X  = ..             S2     X = ...
// S2; S1 gives a different result than S1; S2 => Hazard
```

What hazards can occur in the 5-Stage Pipeline?

- Dependencies on memory operands do not cause hazards because loads/stores are executed in ME stage in program order.
- WAW hazards are not possible because output-register is updated at the end of WB stage in program order. Same argument holds for WAR hazards.
- RAW hazards are real! Exercise: In the table below, which instrs cause hazards?

	Clock⇒	C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	ADD R1,R2,R3	IF	ID	EX	ME	WB				
I2	ADDI R3,R1,#4		IF	ID	EX	ME	WB			
I3	LW R5,0(R1)			IF	ID	EX	ME	WB		
I4	ORI R6,R1,#9				IF	ID	EX	ME	WB	
I5	SUBI R1,R1,R7					IF	ID	EX	ME	WB

RAW Data Hazard Example

	Clock⇒	C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	ADD R1,R2,R3	IF	ID	EX	ME	WB				
I2	ADDI R3,R1,#4		IF	ID	EX	ME	WB			
I3	LW R5,0(R1)			IF	ID	EX	ME	WB		
I4	ORI R6,R1,#9				IF	ID	EX	ME	WB	
I5	SUBI R1,R1,R7					IF	ID	EX	ME	WB

What hazards can occur in the 5-Stage Pipeline?

- I1 updates register R1 at the trailing edge of clock C5.

RAW Data Hazard Example

	Clock⇒	C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	ADD R1,R2,R3	IF	ID	EX	ME	WB				
I2	ADDI R3,R1,#4		IF	ID	EX	ME	WB			
I3	LW R5,0(R1)			IF	ID	EX	ME	WB		
I4	ORI R6,R1,#9				IF	ID	EX	ME	WB	
I5	SUBI R1,R1,R7					IF	ID	EX	ME	WB

What hazards can occur in the 5-Stage Pipeline?

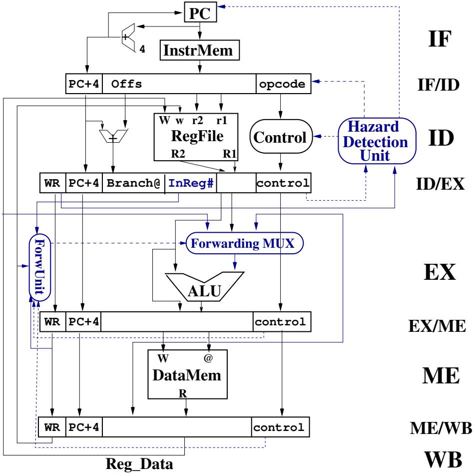
- I1 updates register R1 at the trailing edge of clock C5.
- All other instructions use (read) R1, hence potential hazards.
- I5 reads R1 in ID stage in clock C6, after I1 has updated it in C5, hence RAW dependency is respected (I5 correctly reads the value produced by I1)
- All other instructions I2, I3, and I4 read R1 in ID stage sooner than end of clock C5 ⇒ hence they all read a stale value ⇒ I2, I3, and I4 cause hazards.

Resolving RAW Hazards Generated by ALU Instr

	Clock⇒	C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	ADD R1,R2,R3	IF	ID	EX	ME	WB				
I2	ADDI R3,R1,#4		IF	ID	EX	ME	WB			
I3	LW R5,0(R1)			IF	ID	EX	ME	WB		
I4	ORI R6,R1,#9				IF	ID	EX	ME	WB	
I5	SUBI R1,R1,R7					IF	ID	EX	ME	WB

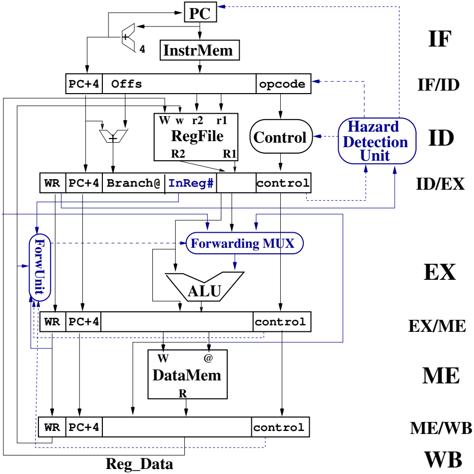
- The hazard by I4 is solved by a small modification in the register file, named **Register Forwarding**: if the same register is read and updated in the same cycle then the updated value is forwarded to the reader (in ID stage). Easy to implement, requires several multiplexers to the register file's ports.
- For the hazards by I2 and I3, observe that the `R1` value is available at the end of C3, while I2 and I3 uses it at the beginning of C4 and C5, respectively.
- **Full Forwarding**: in addition to register forwarding, the new value (of `R1`) can be forwarded to both ALU inputs
  - from register `EX/ME`, at the beginning of C4, thus solving the hazard of I2.
  - from register `ME/WB` (via `REG_data`), at the beginning of C5, thus solving the hazard of I3.

Implementing Forwarding



- Modifications are shown in blue, dotted lines denote control lines.

Implementing Forwarding



- Modifications are shown in blue, dotted lines denote control lines.
- Operand Forwarding uses one three-way multiplexer to **each** input of ALU, but only one is shown. `ForwardingMUX` selects between:
  (1) the "normal" value of `ID/EX`,
  (2) the ALU value, latched in `EX/ME`,
  (3) the WB value, latched in `REG_data`.
- Forwarding Unit controls the 2 MUX:
  (1) Compares the `WR` fields in `EX/ME` and `ME/WB` with the input-register numbers of the currently instr in EX.
  (2) If match selects `EX/ME` or `ME/WB`,
  (3) Else selects `ID/EX`, no hazard.

Forwarding logic is simple, the major complexity is to bus operand values across pipeline stages.

Resolving RAW Hazard Generated by a Load

	Clock⇒	C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	LW R1,0(R3)	IF	ID	EX	ME	WB				
I2	ADDI R3,R1,#4		IF	ID	EX	ME	WB			
I3	LW R5,0(R1)			IF	ID	EX	ME	WB		
I4	ORI R6,R1,#9				IF	ID	EX	ME	WB	
I5	SUBI R1,R1,R7					IF	ID	EX	ME	WB

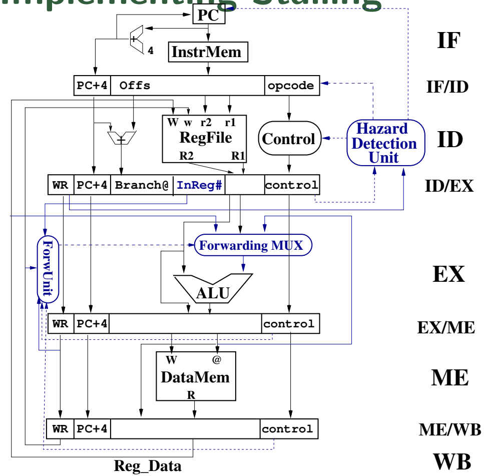
Resolving RAW Hazard Generated by a Load

	Clock⇒	C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	LW R1,0(R3)	IF	ID	EX	ME	WB				
I2	ADDI R3,R1,#4		IF	ID	EX	ME	WB			
I3	LW R5,0(R1)			IF	ID	EX	ME	WB		
I4	ORI R6,R1,#9				IF	ID	EX	ME	WB	
I5	SUBI R1,R1,R7					IF	ID	EX	ME	WB

- The new value (of the load to R1) is available at the end of C4 (end of ME).
- Hazards of I3 and I4 are taken care by register-to-register forwarding,
- It is not possible to forward the new value in time for I2:
  - because I2 needs it at the beginning of C4 in EX, during which the load accesses memory!
  - This hazard need to be detected and I2 (and all following instructions) must be stalled as below ⇒ 1 cycle is lost.

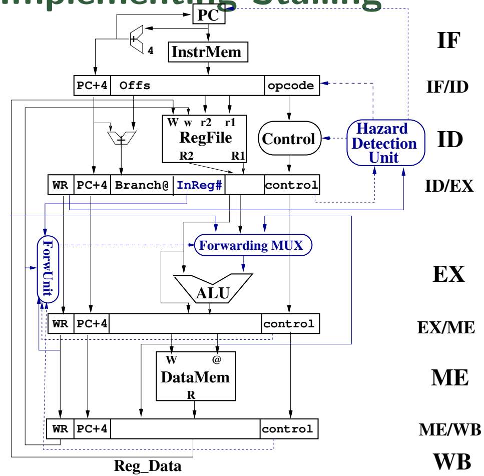
	Clock⇒	C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	LW R1,0(R3)	IF	ID	EX	ME	WB				
I2	ADDI R3,R1,#4		IF	ID	ID	EX	ME	WB		
I3	LW R5,0(R1)			IF	IF	ID	EX	ME	WB	
I4	ORI R6,R1,#9					IF	ID	EX	ME	WB
I5	SUBI R1,R1,R7						IF	ID	EX	ME

Implementing Stalling



- Modifications are shown in blue, dotted lines denote control lines.

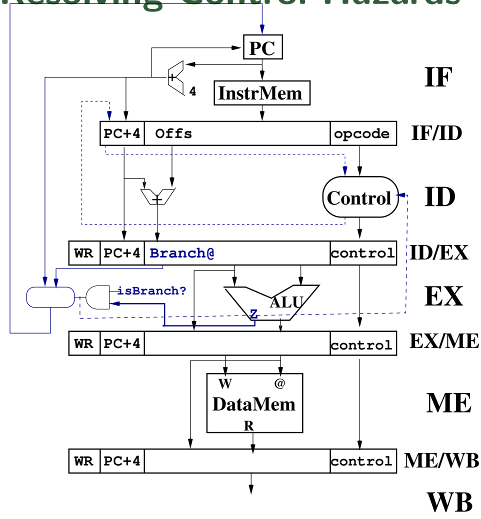
Implementing Stalling



- Modifications are shown in blue, dotted lines denote control lines.
- Hazard Detection Unit (HDU) stalls the pipeline whenever a load is immediately followed by a dependent instr,
- It checks whether the instruction in EX is a load whose destination register is the same as one of the input registers of the next instruction, i.e., the one in ID.
- If so then HDU stalls the IF and ID stages and propagates a NOOP to EX stage. Accomplished by disabling the clock of PC, IF and ID.

Stalling is simpler than forwarding, since no operand value is bussed, but also less efficient, since one clock is lost.

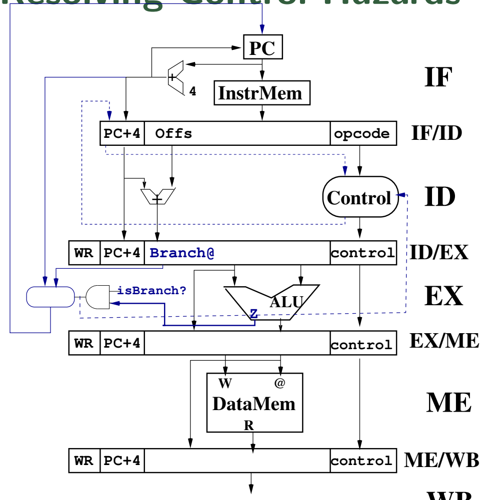
Resolving Control Hazards



- A branch is assumed not taken until the end of EX, when its condition was evaluated by ALU. (Target address is computed in ID.) If ALU's Z bit is set ⇒ branch taken by performing 2 actions:

Modifications are colored blue.

Resolving Control Hazards



- A branch is assumed not taken until the end of EX, when its condition was evaluated by ALU. (Target address is computed in ID.) If ALU's Z bit is set ⇒ branch taken by performing 2 actions:

1. Target address is latched into PC at the end of cycle. If Z=1 (and instr in EX is a branch) then MUX feeding into PC selects the branch target.
2. IF and ID stages must be flushed. This is done by Control when its input is 1, by zeroing out the control fields.

- Jumps with absolute addresses are taken at the end of IF.
- Indirect jumps (JR) are taken at the end of ID, because register containing the target address must be fetched ⇒ instr in IF must be flushed.

Modifications are colored blue.

Two cycles are lost on a taken branch, and one on an indirect jump.

Structural Hazards & Precise Exceptions

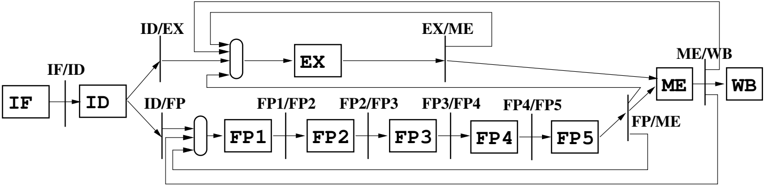
- 5-stage pipeline exhibits NO structural hazards! (conflicts to shared resources).

- Precise exceptions may be triggered in all stages but WB, e.g., page fault in IF/ME, undefined instr in ID, and arithmetic overflow in EX. Whenever they occur:
  1. faulting & following instructions are squashed,
  2. all instructions preceding the faulty one must complete,
  3. execution of exception handler must start.

- However, taking the exception in the same cycle it occurs is very complex:
  1. to-be-flushed stages depend on the stage in which exception occurred,
  2. multiple exceptions may occur in the same cycle in different stages, and
  3. exceptions must be taken in program order, e.g., page faults occurring in IF, but later, a preceding instruction in EX causes another exception.

- A Radical Solution is to flag an exception but keep it silent until instr reaches WB. The first exception on an instr is recorded in an exception-status register (ESR) and the instr is NOOPed. (ESR is carried through the whole pipeline.) Exceptions are taken in WB stage, hence in program order.
- One issue is that a store in ME must be disabled if the preceding instr takes an exception $\Rightarrow$ requires additional hardware.

Out-Of-Order Instruction Completion: Data Hazards

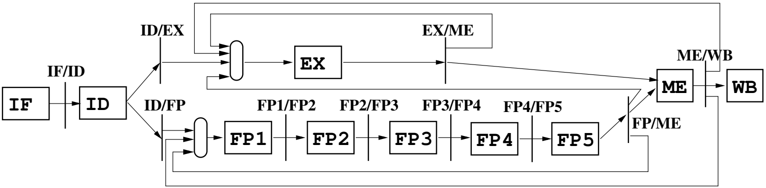


- Based on instr's opcode the decoder sends an instr either on the integer (load/store/ALU/branches) or floating-point pipeline, but all instr go through ME and WB stages. Machine has two separate register files, integer & float.

- Data Hazards:

1 Forwarding is more complicated

Out-Of-Order Instruction Completion: Data Hazards

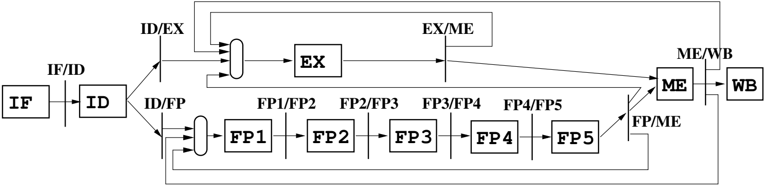


- Based on instr's opcode the decoder sends an instr either on the integer (load/store/ALU/branches) or floating-point pipeline, but all instr go through ME and WB stages. Machine has two separate register files, integer & float.

- Data Hazards:

1. Forwarding is more complicated because float stores need the values of both an integer address and a float register ⇒ a forwarding path was added from `FP/ME` to the input of EX.
2. Hazard Detection Unit (HDU) is more complicated

Out-Of-Order Instruction Completion: Data Hazards

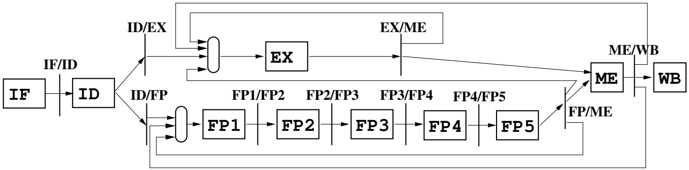


- Based on instr's opcode the decoder sends an instr either on the integer (load/store/ALU/branches) or floating-point pipeline, but all instr go through ME and WB stages. Machine has two separate register files, integer & float.

- Data Hazards:

1. Forwarding is more complicated because float stores need the values of both an integer address and a float register ⇒ a forwarding path was added from `FP/ME` to the input of EX.
2. Hazard Detection Unit (HDU) is more complicated because both RAW and WAW hazards are possible. HDU must stall instrs in ID stage if they have register dependencies with a preceding instruction ⇒ source & destination registers of an instr in ID are checked against ALL destination registers of instrs in execution.
3. Data hazards on memory operands NOT possible because all instructions move through ME in program order.

Metrics: Latency of Operation & Initiation Interval



- **Latency of Operation or Operation Latency** of an instruction is the max # of clocks that the next instruction has to wait (in ID) to avoid RAW/WAW hazards.
- For the pipeline above: it is 1 for Load, and for the rest it is (# of clocks of EX - 1) because closest fwd is at the end of EX.
- Latency of FP unit is large (4) ⇒ stalls are more frequent than in 5-stage pipeline:

	Clock⇒	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
I1	L.D F4,0(R2)	IF	ID	EX	ME	WB						
I2	MULT.D F0,F4,F6		IF	ID	ID	FP1	FP2	FP3	FP4	FP5	ME	WB
I3	S.D F0,0(R2)			IF	IF	ID	ID	ID	ID	ID	EX	ME

- **Initiation Interval** is the min # of cycles between issuing 2 instrs of the same type to execution units.
- In our case is 1 because both integer and float units are fully pipelined, and linear. If float unit would not be pipelined, it would be 5. For dynamically scheduled pipelines it might be variable, i.e., depending on what instructions were previously issued.

Control & Structural Hazards

- Control Hazards solved similarly as in static, 5-stage pipeline.
- Structural Hazards: separate integer and float register files, BUT a float load/store may reach WB stage in the same cycle as a preceding float arithmetic instr ⇒ structural hazard on the write port of FP register file. This is possible because FP instrs in ME use a bus that bypasses memory ≠ from the bus of integer instrs.

	Clock⇒	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
I1	ADD.D F2,F4,F2	IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB	
I2	ADD.D F6,F4,F6		IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB
I3	L.D F8, 0(R1)			IF	ID	EX	ME	WB			
I4	L.D F10,0(R2)				IF	ID	EX	ME	WB		
I5	L.D F14,0(R3)					IF	ID	EX	ME	WB	

Control & Structural Hazards

- Control Hazards solved similarly as in static, 5-stage pipeline.
- Structural Hazards: separate integer and float register files, BUT a float load/store may reach WB stage in the same cycle as a preceding float arithmetic instr ⇒ structural hazard on the write port of FP register file. This is possible because FP instrs in ME use a bus that bypasses memory ≠ from the bus of integer instrs.

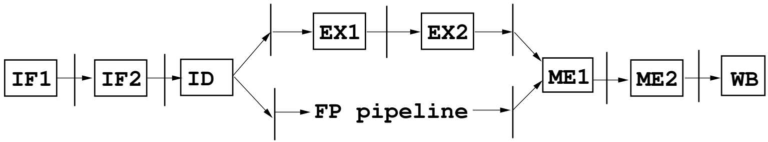
	Clock⇒	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
I1	ADD.D F2,F4,F2	IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB	
I2	ADD.D F6,F4,F6		IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB
I3	L.D F8, 0(R1)			IF	ID	EX	ME	WB			
I4	L.D F10,0(R2)				IF	ID	EX	ME	WB		
I5	L.D F14,0(R3)					IF	ID	EX	ME	WB	

- I1 and I5 being both in ME at C8 is Not a problem because I1 does not access memory. But in the next clock both instrs write the FP register file.
- Can be solved by stalling one of the instrs in ID, or by providing two write ports to the floating point register file.

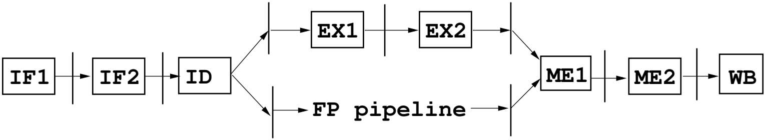
Precise Exceptions in OoO Linear Pipelines

- Major Drawback of OoO pipelines is that precise exceptions are hard to implement, e.g., because instrs reach WB stage out of order.
- Sometimes not implemented: hardware signals the software handler that an exception happened around some PC counter. This is not possible for page faults and strict FP standards (IEEE).
- Conservative technique of stalling in ID until all previous instrs are free of exceptions may stifle pipelining.
- Architecture can be modified to force in-order traversal of WB stage, hence exception can be taken in program order as in static 5-stage pipeline, but at the cost of additional forwarding, and in addition a store can be issued only when all preceding instr are certified free of exceptions.

Superpipelined CPU

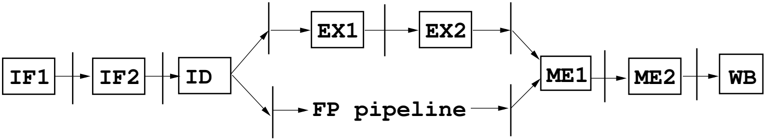


Superpipelined CPU



- In a given technology, some of the 5 stages have more delays than others. For example I-cache access in IF may take twice as decoding. Same with ME, because virtual-mem address translation in TLB is done before accessing the data cache.
- CPU is superpipelined when one of the 5 stages is further pipelined.
- It is also possible to pipeline every single stage (of the 5) to increase throughput.
- *Superpipelines are clocked faster than the worst-case delay of the bottleneck stage of the 5-stage pipeline*. Cycle time decreases, but CPI increases ⇒ no free lunch!
- For example, branches are executed in EX1 and have penalty

Superpipelined CPU

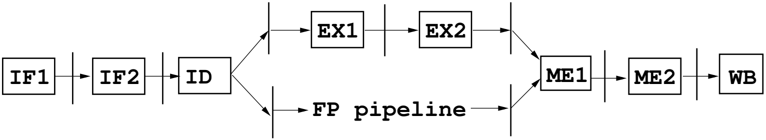


- In a given technology, some of the 5 stages have more delays than others. For example I-cache access in IF may take twice as decoding. Same with ME, because virtual-mem address translation in TLB is done before accessing the data cache.
- CPU is superpipelined when one of the 5 stages is further pipelined.
- It is also possible to pipeline every single stage (of the 5) to increase throughput.

- *Superpipelines are clocked faster than the worst-case delay of the bottleneck stage of the 5-stage pipeline*. Cycle time decreases, but CPI increases $\Rightarrow$ no free lunch!

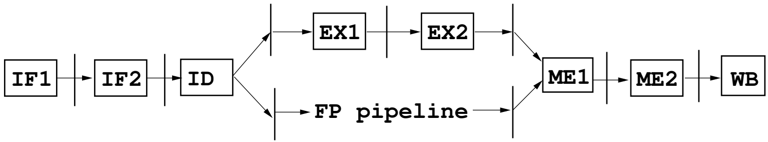
- For example, branches are executed in EX1 and have penalty 3.
- Latency (of operation) of loads is

Superpipelined CPU



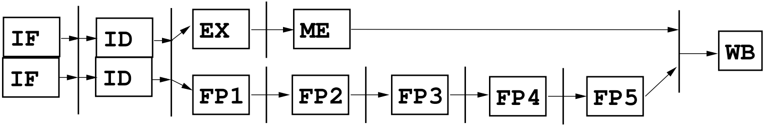
- In a given technology, some of the 5 stages have more delays than others. For example I-cache access in IF may take twice as decoding. Same with ME, because virtual-mem address translation in TLB is done before accessing the data cache.
- CPU is superpipelined when one of the 5 stages is further pipelined.
- It is also possible to pipeline every single stage (of the 5) to increase throughput.
- *Superpipelines are clocked faster than the worst-case delay of the bottleneck stage of the 5-stage pipeline*. Cycle time decreases, but CPI increases $\Rightarrow$ no free lunch!
- For example, branches are executed in EX1 and have penalty 3.
- Latency (of operation) of loads is 3.
- Latency of register-to-register instructions is

Superpipelined CPU

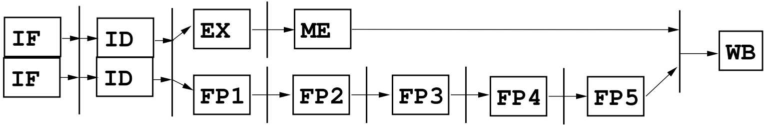


- In a given technology, some of the 5 stages have more delays than others. For example I-cache access in IF may take twice as decoding. Same with ME, because virtual-mem address translation in TLB is done before accessing the data cache.
- CPU is superpipelined when one of the 5 stages is further pipelined.
- It is also possible to pipeline every single stage (of the 5) to increase throughput.
- *Superpipelines are clocked faster than the worst-case delay of the bottleneck stage of the 5-stage pipeline*. Cycle time decreases, but CPI increases ⇒ no free lunch!
- For example, branches are executed in EX1 and have penalty 3.
- Latency (of operation) of loads is 3.
- Latency of register-to-register instructions is 1 (# of EX stages-1).

Superscalar CPU



Superscalar CPU



- More than one instruction is fetched, decoded and issued in each cycle:
- works well on workloads with a good mix of integer and float operations.
- With the above arch, up to two instrs (one int and one float) can proceed to execution once they are free of hazards $\Rightarrow$ CPI < 1

- Supporting precise exceptions is difficult.
- Difficult to build static superscalars wider than 2 ways (int-float),
- e.g., integer pipeline could be split into three pipelines (integer ALU, branches, memory) but such an architecture was never built!

In practice superscalars may be superpipelined as well!

Branch Prediction

Hardwired Prediction is the most static algorithm and out of compiler's control.

Branch Prediction

Hardwired Prediction is the most static algorithm and out of compiler's control.

- always predict branch untaken $\Rightarrow$ no penalty when branch is untaken, but loop backedge misspredicted most of the time.
- predict based on opcode & direction/size of the address offset of the branch instr

1. BNEZ/BEZ is predicted untaken/taken
2. backward branches predicted as taken and forward as untaken
3. branches with large offsets predicted as taken

- Prediction alg. is made at μarchitecture design time based on a mix of benchmarks.

Compiler-Guided Prediction

Branch Prediction

Hardwired Prediction is the most static algorithm and out of compiler's control.

- always predict branch untaken ⇒ no penalty when branch is untaken, but loop backedge misspredicted most of the time.
- predict based on opcode & direction/size of the address offset of the branch instr

1. `BNEZ/BEZ` is predicted untaken/taken
2. backward branches predicted as taken and forward as untaken
3. branches with large offsets predicted as taken

- Prediction alg. is made at μarchitecture design time based on a mix of benchmarks.

Compiler-Guided Prediction

- Uses code profiling to determine the best prediction for each static branch in the code & is communicated to hardware by one extra bit in the branch instr format.
- Note that if a branch is only taken for the first half of prg exec, and untaken for the second half, the compiler will predict it biased (50-50).

For the studied architectures, target address is known only at the end of ID ⇒ instrs in earlier stages must be flushed if branch is predicted taken, even when prediction is correct!

Strength & Weaknesses of Static Pipelines

The studied (static) pipelines schedule instrs in the strict order dictated by compiler (while dynamic archs reorder instrs at runtime to optimize delays/stalls). **Static Pipelines:**

+ are simple ⇒ (1) better clock rate, (2) predictable performance, (3) consume less power since dynamic activities are migrated to the compiler.
- weak on dynamic events such as conditional branches, exceptions, cache misses. For example, the only way to deal with cache misses is to freeze the processor, refill the cache, & replay cycle ⇒ only one mem access can proceed at any one time ⇐ Alleviated by core multi-threading.
- Static instr scheduling is limited by lack of dynamic info, such as memory addresses, e.g., do `a[b[i]]` and `a[b[i+1]]` refer to the same element of `a`?
- Precise exceptions are hard to handle efficiently if OoO instr completion is allowed.
- The design of static pipelines might not scale well, i.e., as pipelines becoming deeper and wider (instr issuing), latencies ↑ and the compiler might not solve them efficiently.

- Static pipelines favored in embedded systems (simplicity),
- Dynamic pipelines, studied towards the end of the course, are/were used in general-purpose environments,
- Trend might reverse due to the advent of core multi-threading.

Core Multi-Threading Overview

Operating System:

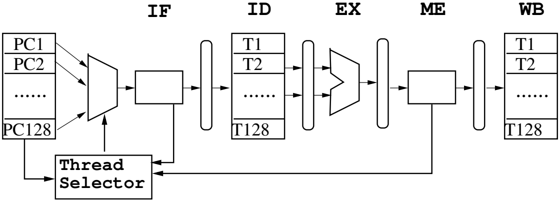
- Resources are precious & expensive $\Rightarrow$ better utilized by time multiplexing (sharing) them over a number of concurrent processes, $\Rightarrow \uparrow$ process throughput.
- An active process is running or ready or blocked on a long latency IO (page fault).
- `running` $\rightarrow$ `blocked/ready` and `ready` $\rightarrow$ `running` transitions involve context switching (100s of cycles overhead). Implemented entirely in software (flexibility).

Core Multi-Threading

- Same idea, but hides much smaller latencies, e.g., level-1/2/3 cache misses, TLB misses, long-latency instrs (co-processor), unsuccessful thread synch, etc.
- Motivated by the ever increasing memory wall & underutilization of functional units (1990s), requires little to no modification of Operating System.
- Each thread runs in a *hardware thread context* (HTC): a set of hwd resources supporting execution and switching.
- A temporarily suspended thread, i.e., not running from hwd view but running from kernel's point of view, is either
  1. *blocked*, i.e., waiting for a pending event to complete, or
  2. *ready*, i.e., HTC allocated but waiting for execution bandwidth.

Barrel Processor or Extreme Core Multi-Threading

Given many enough HTCs and active threads ⇒ No Two Instructions of the Same Thread Are In The Pipeline in the Same Time.



Barrel Processor or Extreme Core Multi-Threading

Given many enough HTCs and active threads ⇒ No Two Instructions of the Same Thread Are In The Pipeline in the Same Time.

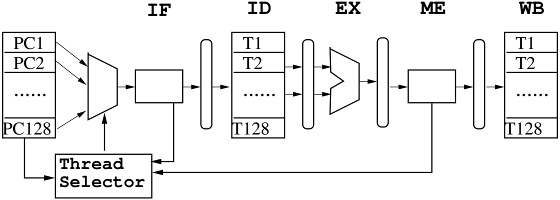


Figure shows a 5-stage pipeline which 128 HTCs, which can hide long mem-access latencies using no or very large and slow caches without ever starving the pipeline.

Barrel Processor or Extreme Core Multi-Threading

Given many enough HTCs and active threads ⇒ No Two Instructions of the Same Thread Are In The Pipeline in the Same Time.

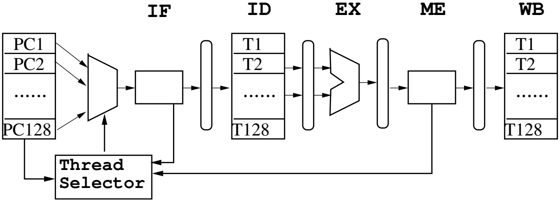


Figure shows a 5-stage pipeline which 128 HTCs, which can hide long mem-access latencies using no or very large and slow caches without ever starving the pipeline.

- Data/Control/Structural Hazards

Barrel Processor or Extreme Core Multi-Threading

Given many enough HTCs and active threads ⇒ No Two Instructions of the Same Thread Are In The Pipeline in the Same Time.

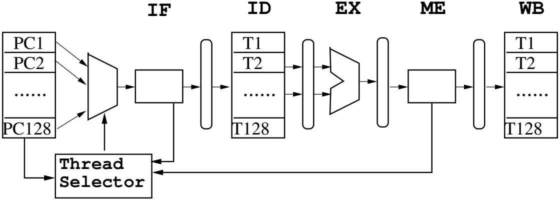


Figure shows a 5-stage pipeline which 128 HTCs, which can hide long mem-access latencies using no or very large and slow caches without ever starving the pipeline.

- Data/Control/Structural Hazards cannot occur because of the assumption; Precise Exceptions

Barrel Processor or Extreme Core Multi-Threading

Given many enough HTCs and active threads $\Rightarrow$ No Two Instructions of the Same Thread Are In The Pipeline in the Same Time.

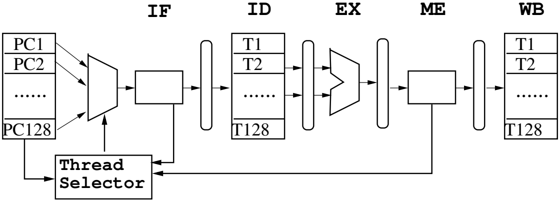


Figure shows a 5-stage pipeline which 128 HTCs, which can hide long mem-access latencies using no or very large and slow caches without ever starving the pipeline.

- Data/Control/Structural Hazards cannot occur because of the assumption; Precise Exceptions are respected because of the assumption. Problems are:

Barrel Processor or Extreme Core Multi-Threading

Given many enough HTCs and active threads ⇒ No Two Instructions of the Same Thread Are In The Pipeline in the Same Time.

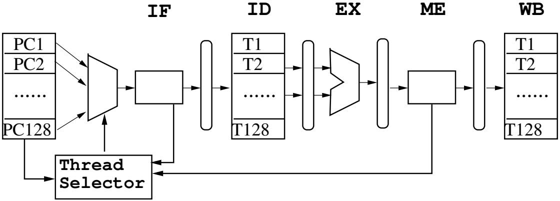


Figure shows a 5-stage pipeline which 128 HTCs, which can hide long mem-access latencies using no or very large and slow caches without ever starving the pipeline.

- Data/Control/Structural Hazards cannot occur because of the assumption;

  Precise Exceptions are respected because of the assumption. Problems are:
  1. with current technology 1000s of threads would be needed in workload and such hardware replication would be prohibitive,
  2. memory wall was getting larger and larger,
  3. too different from legacy code
  4. if num running threads < 5 ⇒ NOOPs inserted ⇒ slow sequential execution.

Barrel Processor or Extreme Core Multi-Threading

Given many enough HTCs and active threads ⇒ No Two Instructions of the Same Thread Are In The Pipeline in the Same Time.

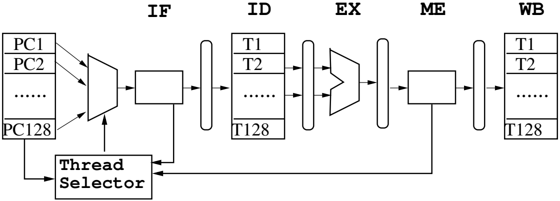


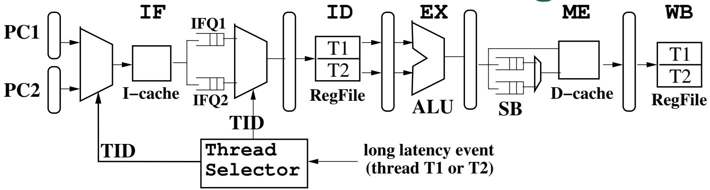
Figure shows a 5-stage pipeline which 128 HTCs, which can hide long mem-access latencies using no or very large and slow caches without ever starving the pipeline.

- Data/Control/Structural Hazards cannot occur because of the assumption;

  Precise Exceptions are respected because of the assumption. Problems are:
  1. with current technology 1000s of threads would be needed in workload and such hardware replication would be prohibitive,
  2. memory wall was getting larger and larger,
  3. too different from legacy code
  4. if num running threads < 5 ⇒ NOOPs inserted ⇒ slow sequential execution.
- CDC 6600 (Control Data Corporation) in 1960s: 32-way multi-threading (MT).
- Later in 1980s, Deneclor's HEP: 16-way MT, 8-stage pipeline, no cache.
- Denclor's TERA contains 256 Horizon Processors, each a 128-way MT, 3-stage pipeline. PC & register file replicated. No data cache ⇒ no cache coherence.

  Estimated that the pipeline is fully utilized even if instrs takes 80 cycles to execute!

Interleaved Core Multi-Threading



Several threads run simultaneously. Processor fetches, decodes, and schedules instrs from ≠ threads in consec cycles.

A 2-way MT must have 2 PCs, 2 sets of regs and interrupt flags, 2 page table base regs, BUT shares L1/2 caches & pipeline.

Other resources might be replicated, e.g., branch prediction hwd and TLB, to avoid prediction interferences between tasks.

Each instr carries extra bits that identifies its thread id ⇒ thread-aware data forwd, stage flushing (exception & branches). e.g., on a L1 data-cache miss of `T1`, ME raises a low-level hwd exception that directs the flushing and insertion in IFQ1 of the instrs of T1 in ID, EX and ME.

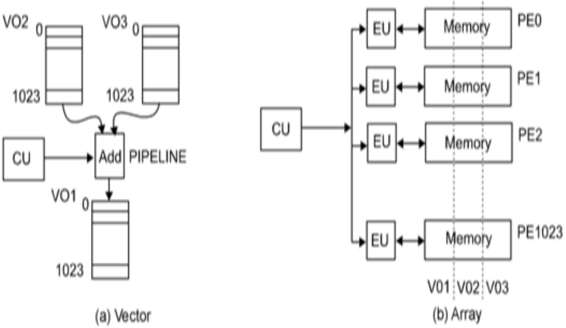
Seq exec same speed as 5-stage pipeline.
Multiple running thds ⇒ reduced latencies:
[1] 5 running thds eliminate all hazards.
[2] 2 running thds eliminate the bubble of load followed by dependent instr.

SunSparc T1&T2 (shown in Figure) targets workloads with lots of thds, but with little to no FP operations, in which the runtime is dominated by cache misses. Decoupling the IF stage via 2 IF-queues smooths out instr delivery in the case of cache misses. Branches are statically predicted by instr bit set by compiler. SunSparc T1 has 8 cores, 4-way MT, and SunSparc T2 has 8 cores, 8-way MT each.

Vector and Array Processors

Assume vectors of 64 words, and that ADD has 10 stages.
Total Execution Time:
$T_{exe} = 10 + 63 = 9 + 64$

In general:
$T_{exe} = T_{start} + N$,
where N is vector's length.



After vector's startup time, i.e., the time to get the first result, scalar results are computed one per clock:

T_{vector} `= Vector Length + Startup Time`

Load/Store Vector Architecture

- vector registers + scalar registers
  - each vector register can hold consec vector components fetched from memory, e.g, 8 vector registers of 64 components each
  - 1 READ and 1 WRITE port per vector register, connected to function units via crossbar interconnect (all registers to all units).
- Vector functional units are specialized and deeply pipelined, and act on operands in vector registers

HL Code	Machine Code	Comments
`for(i=0; i<2048; i++)` `  Y[i] = a*X[i] + Y[i]`	``` ADDI R6,R0,#1```	`//sets R6 to 1 (the stride)`
	`LOOP:`	`//load slice of vector X of`
	`L.V    V1,0(R1),R6`	`//  base 0(R1) & stride R6`
	`MUL.V  V2,V1,F0`	`//multiply X with scalar in F0`
	`L.V    V3,0(R2),R6`	`//load slice of vector Y`
	`ADD.V  V4,V2,V3`	`//add vector slices`
	`S.V    V4,0(R2),R6`	`//store slice of Y in memory`
	`SUBBI R1,R1,#512`	`//jump to next slices,`
	`SUBBI R2,R2,#512`	`//i.e., 8*64`
	`SUBBI R3,R3,#64`	
	`BNEZ  R3,LOOP`	

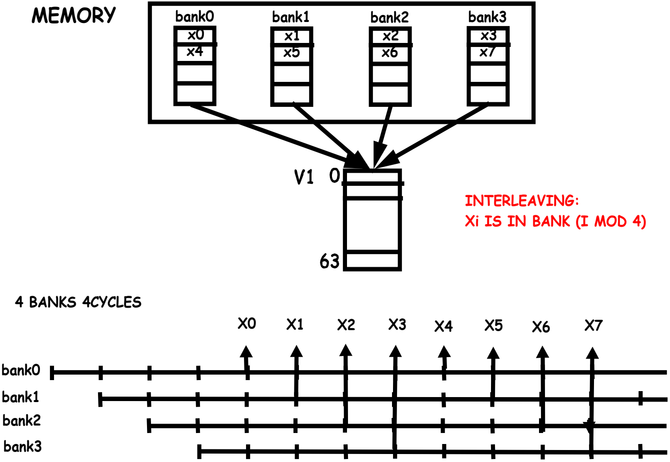
Vector – Memory System

- Memory Access Pattern known at decode time for entire vector
  - memory is interleaved, no need for caches,
  - vector load/store units bring data from memory to registers.
- Load/Store Units can also be seen as pipelines
  - Startup time: is the time to bring the first component (much longer than for functional units),
  - banks are started one after another
  - If the number of banks $>>$ memory cycle time Then there can be no conflicts and results come out one per clock,
  - vectors address with a stride are stored in consecutive locations of vector register.

Vector – Memory Organization

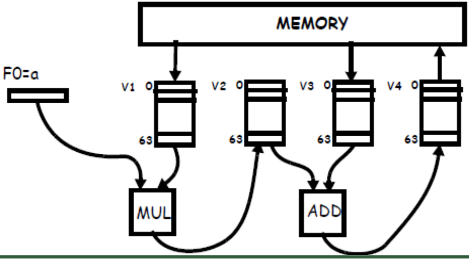
T_{load} = Vector Length + Startup Time (TimeToGet x_0 - 1).

Memory is heavily interleaved: hundreds of memory modules:



Chaining and Parallel Execution

- Independent Vector Instructions can be executed in parallel if functional units are available, but more important
- They can be chained when there are dependencies. Consider `Y = aX + Y`, where vector length is N:
- Execution Time (One Vct Op at a time): `2*startup(load) + startup(multv)+startup(addv)+startup(store) + 5*N`
- Execution Time (Chaining and Parallel): `startup(load) + startup(multv) + startup(addv) + startup(store) + N` (assuming the `loads` go in parallel)




```
L.D     F0,0(R4),R6
L.V     V1,0(R1),R6
MUL.V   V2, V1, F0
L.V     V3,0(R2),R6
ADD.V   V4, V2, V3
S.V     0(R2),V4,R6
```

Control Flow & Indirect Accesses

For conditional statement use Predicated Instructions

Fortran Code	Machine Code with Predicated Instructions

`DO i = 1, 64, 1`
`  IF (A(i) .NE. 0) THEN`
`    A(i) = A(i) - B(i)`
`  ENDIF`
`ENDDO`
	`// Use a Bit-Vector Mask Register VM:`
	`// Assuming A is in V1 and 0 in F0`
	`SNEVS V1, F0   //sets VM(i) if V1(i) ≠F0`
	`SUB.V V1,V1,V2 //subtract under Vector Mask`

Scatter/Gather Operations:

- Many scientific computations use sparse matrices
- Most components are zeros, but under irregular pattern
- Compress a sparse vector `A` into vectors of non-zero elements: `A[1:1000]` $\Rightarrow$ `A*[1:9]`, `K[1:9]`, where `A*` are the non-zero elements of `A` and `K` holds their original indexes. Code example:
- ```
  DO i = 1, N
     A(K(i)) = A(K(i)) + C(M(i))
  ENDDO
  ```
  ```

Memory Scatter/Gather Operations

- Memory Locations of Vector Components are spread out in memory
- $\Rightarrow$ use scatter and gather instructions
  - **Gather** is an instruction loading a set of indirect-indexed addresses into consecutive vector-register locations
  - **Scatter** is a store instruction from a vector register to indirectly indexed addresses.

```
L.V   Vk,0(R1),R6  //loads vector K in Register Vk
LI.V  Va,Vk,0(R2)  //gather: loads A(K(i)) into Va
SI.V  Va,0(R2),Vk  //scatter: stores A(K(i))
```