Lecture outline

- Complex addressing modes.
- Complex data hazards.
- Addressing cache latency
- Long (and different) latency instructions.
- Superpipelining
- Dealing with interrupts.
Complex Addressing Modes

• Auto-increment causes register change during instruction execution
  – Interrupts? Need to restore register state
  – Adds WAR and WAW hazards since writes are no longer restricted to the last stage
  – More than one destination register for an instruction (two registers change value) ⇒ increase number of register file ports.

• Deferred addressing modes:
  – Long decode process spanning across multiple pipelined stages.
  – Need to access memory in order to supply source operands ⇒ may have page faults (or other memory related exceptions).
Complex Data Hazards

• **Condition Codes:**
  – are written as a side effect during execution.
  – are read by various instructions (e.g. branches)
  – ⇒ RAW, WAW and WAR on condition codes.
  – Solution should be similar to the way it is solved with general-purpose registers, i.e., treat the problem as data hazards.

• **Memory ambiguity:** data hazards occurring between data items stored in memory (rather than registers).
  – Can only be detected when effective addresses are known.
  – As long as effective addresses are not computed, need to adopt the conservative approach.
  – Irrelevant in the basic DLX since both memory reads and memory writes occur in the same pipeline stage.
Overcoming Cache Latency

IF:  PC selection; initiate cache access
IS:  2nd half; complete cache access
RF: ID; Reg. fetch; hazard check;
    + Instruction cache hit detection
EX:  Ex; Eff. Addr.; Branch cond.+target
     (unless done earlier)
DF, DS: 1st, 2nd half of data access
TC:  Tag check for data access
WB

Benefit:
- Shorter clock cycle → higher throughput

Negative implications:
- More forwarding
- Longer (in clock cycles) Load and Branch delays
Long (and Variable) Latency Instructions

- Usually floating point
- **Floating point execution:**
  - multiple units
  - partly pipelined
    - latency
    - initiation rate
- **Instruction issue:**
  ID → EX
- **Instruction commit:**
  guaranteed to complete
- **Hazard checking:**
  just before issue
Long Latency Instructions

• Long latency instructions
  – Require multiple cycles in EX
  – EX may itself be pipelined (or non-pipelined)
  – Regardless of pipelining of EX, having more than one
    ALU and instructions with different execution times may
    cause out-of-order completion of instructions

• Most Floating Point instructions are long
  latency instructions.

• Divide, Square-root take 10X to 30X longer than
  Add
Long (variable) Length Instr. Complications

- Execution units (e.g., Divide) that are not fully pipelined can present structural hazards.
  **Solution:** stall until unit is available

- May require multiple writes to reg. file in same cycle (but only one on average!)
  **Solution:** additional write ports or stall (usually stall because there is no problem on average); alternative: write buffer.

- WAW and WAR hazards. (But no WAR in DLX/MIPS)
  **Solution:** use additional temporary registers.

- Out of order completion $\Rightarrow$ difficult exception (interrupt) handling.
  Need an ability to undo (next slide)

- Longer latency $\Rightarrow$ stalls cycles for RAW more frequent.
Long-Latency Instr. - Solution Mechanisms

• Main generic problems:
  – out of order completion
  – need to undo

• Approaches for solution:
  – extend “short” pipeline segments to defer change of state, so all instructions take the same time from IF to WB (e.g., “reorder buffer”)
  – limit the flexibility in the use of the pipeline; e.g., effectively at most one FP operation active at any given time.
  – develop the capability to “undo”:
    » history file (old values of registers kept for a while); when there is a problem, reload registers from them.
    » future file (keep new values on the side; when safe, copy them into “true” registers.

Remark: the additional registered are “private”; they are not exposed externally, e.g., to the compiler, and may vary from one implementation to another without compromising compatibility.
Implementation of Structural Hazard

- Write-port structural hazard detection and enforcement:
  - Detection in ID
    » Keep a “sliding window” of write-port reservations
    » In ID, check to see whether the port is going to be available when the instruction is scheduled to need it. If not, stall.
    » Benefit: all tests during ID
    » Cost: shift register and related logic
  - Just-in-time detection
    » Detect hazard at entry to MEM or WB
    » Stall one of the instructions
    » Benefit: have a choice of which instruction to stall (e.g., one from shorter-latency unit or less frequently used unit)
    » Cost: complicated pipeline control.

We will assume detection during ID unless stated otherwise
Implementation of WAW Hazard

• Enforce order:
  – detect upcoming hazard
  – stall later instruction as necessary to avoid hazard

• Nullify effect of earlier instruction
  – detect upcoming hazard
  – change earlier instruction in pipeline so as to turn off write
Dependences between Int and FP Instructions

• Only dependences:
  – FP load and store (carried out by Int. unit)
  – FP-Int register copies

• Benefit:
  – simplicity
  – can have twice as many architectural registers with same number of register-address bits in instruction

• Cost:
  – Extra set of registers
  – moves between the two sets
Before issuing an instruction in ID, check:

- **Structural hazards:**
  - functional unit availability
  - write-port availability

- **RAW hazards:**
  - make sure that source registers are not listed as destination registers of earlier instructions that will not generate the data in time for its use by current instruction

- **WAW hazards:**
  - make sure that if an earlier instruction has the same destination address, it reaches the write stage earlier than the current instruction
Parallelism

- Parallelism: allowing multiple things to occur concurrently in order to increase throughput and possibly reduce latency.

- Granularities of parallelism:
  - different programs or processes executed concurrently on different processors (compiler doesn’t have to know)
  - thread-level parallelism: multiple processors working on same program (programmer and/or compiler must know)
  - Instruction-level parallelism (ILP): different instructions of the same program executed concurrently by functional units of the same processor.
Instruction Level Parallelism (ILP)

- **Definition:** the degree of ILP in a given program is the average number of instructions that can potentially be executed in parallel.

- **Challenge:** detect and exploit ILP
  - multiple instructions executed in parallel
  - CPI<1 is possible

- **Achievable degree of ILP is limited:**
  - Limited machine resources: execution units, memory ports, etc.
  - Data dependences:
    - True-Data dependences.
    - Name (false) dependences.
  - Control dependences.
  - Instruction-fetch bandwidth.
Data Dependences

- **True-data dependence:**
  - e.g.:
    - add r1, r2, r3
    - add r6, r5, r1

- **Output dependence:**
  - e.g.:
    - add r1, r2, r3
    - add r6, r5, r1
    - add r1, r9, r7

- **Anti dependence:**
  - e.g.:
    - add r2, r5, r1
    - add r4, r2, r3
    - add r1, r15, r10

Also termed name dependences
Solutions for Data dependencies

- **Main idea:** the use of “private” registers to store speculative or prematurely-computed data relaxes many precedence constraints.

- **Register renaming:**
  - Architectural registers = virtual registers
  - Machine registers = physical register
  - Map virtual register to physical register using Register Alias Table.
  - Can solve all name dependences as long as there are available unallocated registers.

- **True data dependences:**
  - Value prediction – attempting to predict the result of instructions and “feeding” the dependent instructions with the predicted value.
  - In case of incorrect prediction, dependent instructions need to be re-executed.
Control Dependences

- Appear when the execution of instructions depends on the outcome of a branch (decision + target address), and limit the degree of ILP

- Normally, can only execute concurrently instructions from the same basic block (between consecutive branches)

- Goal: boost parallelism by searching for executable instructions from various basic blocks (average basic block size is 4-5 instructions), thereby breaking the control flow boundaries.

- Solutions:
  - Various branch prediction techniques (BTB, 2-level BTB, RSB etc.).
  - Speculative execution:
    » execute instructions according to the branch prediction.
    » Instructions are executed before branch target is known.
    » In case of branch misprediction, need to clean the pipeline.
    » Challenge: prevent speculatively executed instruction from changing the architectural machine state (registers, flags, memory etc.).
The order of instructions stored in memory does not always correspond to the execution order.

A program consists of basic blocks:
- A basic block – a continuous sequence of instruction between branches.

E.g., various basic block sequences may be executed, like, ABC, WEH, AZG, ...

Requires multiple ports to memory/l-cache.

Trace cache – stores trace of instructions in a cache line:
- Complicates the instruction fetch mechanism.
- Redundancy of code in traces, e.g.: ABC, ABD, ABE, ...
- Partial matching: programs executed ABC but found ABD.

Compiler based solutions – various basic block enlargement techniques like trace scheduling attempt to combine small basic blocks to one contiguous large basic block.
A program’s dataflow graph \((DFG)\) is a directed graph \(G(V,S)\):

- Each node, \(V\), represents a single instruction. A node number represents the appearance order of the instruction in the trace of the program.
- Each arc, \(S_{ij}\), represents a true-data dependence between nodes \(i\) and \(j\) \((i,j \in V)\).

The DFG also reveals which instructions can be executed in parallel, given that all other dependences are resolved.
Handling Interrupts

- **Interrupt**: a change in the flow of a program that is not due to a jump, branch or procedure call.

- **A variety of interrupts**: I/O Devices, System calls, Breakpoints, over/underflow, FP anomaly, Page fault, Misaligned mem access, Mem protection violation, Hardware malfunction, Power failure, etc.

- **With 5 instructions executing in a 5-stage pipeline**:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Potential interrupts occurring</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on instruction fetch; misaligned memory access; memory-protection violation</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined or illegal opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic interrupt</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data fetch; misaligned memory access; memory-protection violation</td>
</tr>
</tbody>
</table>

- **Interrupt classification**:
  - can interrupt be masked?
  - must it be handled during an instruction or can it wait for its completion?
  - does the program abort or continue after the interrupt?

**Most challenging**: unmaskable, during instruction, must resume.
Handling Interrupts (cont.)

• “Precise exceptions (interrupts)”: Interrupts must be handled in a way that guarantees the same result as with a CPU that executes one instruction at a time (no pipeline).

• Challenges with a multi-stage pipeline and several active instructions:
  – How to stop the pipeline?
  – How to restart the pipeline?
  – Who caused the interrupt?

• Simultaneous exceptions in more than one pipeline stage, e.g.:
  – Load with data page fault in MEM stage
  – Add with instruction page fault in IF stage
  – Add fault will happen BEFORE load fault

• Complications with branch delay slots: When the instruction in the branch delay slot causes an exception, what would be the return address?
Approaches to Interrupt Handling

- **Solution #1**
  - Interrupt status vector per instruction
  - Defer check till last stage
  - Expose interrupts in a sequential manner
  - Flush pipeline when jumping to the interrupt handler.

- **Solution #2**
  - Interrupt ASAP
  - Restart everything that is incomplete
  - Idea: as long as all interrupts are handled, order doesn’t really matter
  - ⇒ (formally) Imprecise exceptions, but OK in many cases (e.g., page fault)

- **Problem (especially with FP instructions):** instructions may write results before possibility of interrupt by another instruction is eliminated. Some processors support both modes: use one for debugging
Special Problem: Interrupt in a Branch-Delay Slot

| Target+1 | Branch Target | Branch-delay 2 | Branch-delay 1 | Branch |

• **Problem:**
  – the instruction in the branch delay slot appears after the branch
  – yet, the identity of the next instruction that should be executed depends on the outcome of the branch

• **Solution:** must carry PC values for all instructions that may be affected (No. of branch delay slots +1)
Exception Handling Complication- Example

• Program:
  - DIV.D F0, F2, F4
  - ADD.D F10, F10, F8
  - SUB.D F12, F12, F14
  - No hazards
  - ADD.D and SUB.D complete long before DIV.D
  - Suppose that SUB.D causes a floating point arithmetic exception when ADD.D has completed but DIV.D has not

• Problems:
  - ADD.D has already written to register, so stall cannot help
  - ADD.D destroyed one of its operands, so even software cannot help
Exception Handling Example – cont.

- **Solution 1:**
  - ignore. (Imprecise exception)
  - Have a special “debug” mode to ensure correctness.

- **Solution 2: buffer results until certain**
  - history file: keep old values in temporary registers until certain; use if needed or discard.
  - future file: keep new values in temporary registers until certain; then copy to formal registers; discard if there is a problem.

- **Solution 3:**
  - keep track of all instructions in the execution window
  - resume under software control
  - can get very complicated unless there are restrictions on instruction mix

- **Solution 4: allow issue to proceed only if certain that earlier instructions in the pipe will complete without causing interrupts**
  - later instructions than the interrupt causer are guaranteed not to complete prior to the interrupt
  - the ability of earlier ones to complete is guaranteed
  - requires test at beginning of EX to make sure that instruction will not cause an interrupt