Computer Organization & Architecture

Performance evaluation

- For a non-pipelined processor, execution time, *T*, of a program that has a dynamic instruction count of I_c is:

$$T = I_c \times S / f$$

where *S* is average number of clock cycles it takes to fetch and execute one instruction, and *f* is clock rate in cycles per second.

- For non-pipelined execution, throughput, P_{np}, is:

$$P_{np} = f / S$$

- For a processor that uses five cycles to execute all instructions, if there are no cache misses, $S = 5$

Performance evaluation

- Pipelining improves performance by overlapping execution of successive instructions
  - Increases instruction throughput
  - Individual instruction is still executed in same number of cycles.
  - For five-stage pipeline, each instruction is executed in five cycles
  - But a new instruction can ideally enter the pipeline every cycle.

Performance evaluation

- In absence of stalls, $S = 1$, and ideal throughput with pipelining is:

$$P_p = f$$

- A 5-stage pipeline can potentially increase throughput by factor of five.
  - In general, an n-stage pipeline has the potential to increase throughput n times.

Effects of stalls/penalties

- Operations with longest delay dictate cycle time, and hence clock rate *f*.
- For a processor that has on-chip caches:
  - Memory-access operations have small delay when desired instructions or data are found in cache
- Delay through ALU is likely to be critical parameter.
  - If this delay is 2 ns, then f = 500 MHz
  - The ideal pipelined instruction throughput is

    P_p = 500 MIPS (million instructions per second)

Effects of stalls/penalties

- Consider a processor with operand forwarding in hardware
  - No penalties due to data dependencies, except in case of Load instructions.
- Consider how often a Load instruction is immediately followed by another instruction that uses result of memory access.
  - A one-cycle stall is necessary in such cases.
  - Ideal pipelined execution has $S = 1$, stalls due to such Load instructions have effect of increasing S by an amount δ_{stall}.

Effects of stalls/penalties

- Ex., Assume:
  - Load instructions constitute 25% of dynamic instruction count
  - 40% of Load instructions are followed by dependent instruction.
  - A one-cycle stall is needed in such cases.
  - Increase over the ideal case of S = 1 is

$$\delta_{\text{stall}} = 0.25 \times 0.40 \times 1 = 0.10$$

  - Execution time T is increased by 10 percent
  - Throughput is reduced to $P_p = f / (1 + \delta_{stall}) = f / 1.1 = 0.91f$

Effects of stalls/penalties

- Consider penalties due to mispredicting branches:
  - When both branch decision and branch target address are determined in Decode stage, branch penalty is one cycle.
  - Assume branches constitute 20% of dynamic instruction count, and average prediction accuracy for branch instructions is 90%
  - That is, 10% percent of all branch instructions that are executed incur a one-cycle penalty due to misprediction.
  - Increase in average number of cycles per instruction due to branch penalties is

$$\delta_{branch\_penalty} = 0.20 \times 0.10 \times 1 = 0.02$$

Number of pipeline stages

- An *n*-stage pipeline:
  - May increase instruction throughput by a factor of *n*
  - Should use a large number of stages??
- As number of pipeline stages increases:
  - More instructions being executed concurrently.
  - More potential dependencies between instructions that may lead to pipeline stalls.
  - Branch penalty may be larger than one cycle if a longer pipeline moves the branch decision to a later stage.
- Gain in throughput from increasing value of *n* begins to diminish
  - Cost of a deeper pipeline may not be justified.

Number of pipeline stages

- Another important factor is inherent delay in basic operations performed by processor.
  - Most important is ALU delay.
  - In many processors, cycle time of processor clock is chosen so that one ALU operation can be completed in one cycle.
- Other operations, including accesses to a cache memory:
  - Typically divided into steps that each take about same time as an ALU operation.

Number of pipeline stages

- Further reductions in clock cycle time possible if a pipelined ALU is used.
  - Some recent processor implementations have used twenty or more pipeline stages to aggressively reduce the cycle time.

Superscalar Operations

- Equip processor with multiple execution units
  - Each may be pipelined, to increase processor's ability to handle several instructions in parallel.
  - Several instructions start execution in same clock cycle, but in different execution units
  - The processor is said to use multiple-issue.
  - Processors can achieve instruction execution throughput of more than one instruction per cycle.
  - Known as **superscalar processors**.
- Many modern high-performance processors use this approach.

Superscalar Operations

- To enable multiple-issue execution:
  - Superscalar processor has elaborate fetch unit
  - Fetches two or more instructions per cycle before they are needed
  - Places them in an instruction queue.
- A separate unit, called the dispatch unit:
  - Takes two or more instructions from front of queue
  - Decodes them, and sends them to appropriate execution units.
- At the end of the pipeline, another unit is responsible for writing results into register file.

Superscalar Operations

- A superscalar processor may incorporate two execution units
  - One for arithmetic instructions and another for Load and Store instructions.
  - Arithmetic operations normally require only one cycle, hence first execution unit is simple.
  - Load and Store instructions involve an address calculation for Index mode before each memory access, they have a two-stage pipeline

Superscalar Operations

- Challenges presented by branch instructions and data dependencies can be addressed with additional hardware.
  - Fetch unit handles branch instructions as it determines which instructions to place in queue for dispatching
  - It must determine both branch decision and target for each branch instruction
  - Branch decision may depend on result of an earlier instruction that is either still queued or newly dispatched

Superscalar Operations

- Stalling fetch unit until result is available can significantly reduce throughput
  - Not a desirable approach
  - Better to employ branch prediction

Superscalar Operations

- Since aim is to achieve high throughput
  - Prediction is combined with a technique called **speculative execution.**
  - Subsequent instructions based on an unconfirmed prediction are fetched, dispatched, and possibly executed
  - But labeled as being speculative so that they and their results may be discarded if the prediction is incorrect.

Superscalar Operations

- **Out-of-Order Execution:**
  - Instructions are dispatched in same order as they appear in program.
  - However, their execution may be completed out of order.
- Ex.,

```
Add R2, R3, #100
Load R5, 16(R6)
Subtract R7, R8, R9
Store R10, 24(R11)
```

  - Subtract instruction writes to register R7 in same cycle as Load instruction that was fetched earlier writes to register R5.
  - If memory access for Load instruction requires more than one cycle to complete, execution of Subtract instruction would be completed before Load instruction.

Superscalar Operations

- If an instruction I_{j+1} depends on the result of instruction I_j:
  - Execution of I_{j+1} will be delayed if result is not available when it is needed.
- As long as such dependencies are handled correctly:
  - No reason to delay execution of an unrelated instruction.
- If there is no dependency between a pair of instructions:
  - Order in which execution is completed does not matter
  - Results of execution of instructions must be written into destination locations strictly in program order

Superscalar Operations

- To improve performance, an execution unit allowed to execute any instructions whose operands are ready
  - May lead to out-of-order execution of instructions.
  - However, instructions must be completed in program order to allow precise exceptions.
- Can be resolved if execution is allowed to proceed out of order, but results are written into temporary registers.
  - Contents of these registers are later transferred to permanent registers in correct program order.
  - Last step is called **commitment** step, because effect of an instruction cannot be reversed after that point.

Superscalar Operations

- If an instruction causes an exception:
  - Results of any subsequent instructions that have been executed would still be in temporary registers and can be safely discarded.
  - Results that would normally be written to memory would also be buffered temporarily, and can be safely discarded as well.
- A temporary register that is assigned for result of an instruction assumes role of permanent register whose data it is holding.
  - Its contents are forwarded to any subsequent instruction that refers to original permanent register during that period.
  - This technique is called **register renaming**

Superscalar Operations

- When out-of-order execution is allowed:
  - Special control unit, called **commitment unit** is needed to guarantee in-order commitment.
  - Uses a separate queue called **reorder buffer** to determine which instruction(s) should be committed next.
  - Instructions are entered in queue strictly in program order as they are dispatched for execution.
- When instruction reaches head of this queue and execution of that instruction has been completed:
  - Corresponding results are transferred from temporary registers to permanent registers
  - Instruction is removed from queue.
  - All resources assigned to instruction, including temporary registers, are released.
  - Instruction is said to have been **retired** at this point.

Superscalar Operations

- Because an instruction is retired only when it is at the head of the queue:
  - All instructions that were dispatched before it must also have been retired.
  - Hence, instructions may complete execution out of order, but they are retired in program order.

END