**Pipelining**

* Pipelining is an architectural technique for improving speed of programs
  + With 5-stage datapath, instructions are executed one at a time with one stage active in any clock cycle
  + Pipelining allow multiple stages for different instructions to be simultaneously active
* **Pipeline organization**
  + A new instruction enters the pipeline every cycle
  + Information about each instruction is carried through the pipeline as the instructions flow through the stages
    - Stored in interstage buffers – i.e. the interstage registers + more
    - B1 – feeds instruction fetched from fetch stage to decode stage
    - B2 – feeds execute stage; holds:
      * 2 operands from register file
      * Immediate value
      * Incremented PC (as return address)
      * Control signals settings
    - B3 – feeds memory stage; holds:
      * Result from ALU
      * Data to be written to memory
      * Incremented PC (from B2)
    - B4 – feeds writeback stage; holds:
      * Value to be written to reg file
        + Can be result from ALU, result from memory stage, or incremented PC
* Hazard – any condition that requires the stalling of pipeline
  + E.g. consider the instructions:

ADD R2, R3, #100

SUB R9, R2, #30

* + - R2 is written to in stage 5 of I1 (cycle 5)
    - R2 is read from in stage 2 of I2 (cycle 3)
  + There is a data dependency between these instructions – a data hazard is caused
    - In particular, this is a read-after-write (RAW) hazard
  + Pipeline must be stalled so that R2 is read in cycle 6
    - Control circuitry compares source operands of I1 (stored in B2) and dest operand of I2 (stored in B1) when I2 is decoded in cycle 3
    - Recognizes data dependency, stalls I2 by inserting NOPs (no-operation – one clock cycle of idle time; a.k.a. a bubble)
    - I2 is held in B1 while I1 is carried through the pipeline

Clock 1 2 3 4 5 6 7 8

I1 F D C M W

I2 F D (stall) C M W

* Operand forwarding – dependencies can be handled w/o stalling the pipeline
  + Although R2 is written to in cycle 5, the result of I1 is available at the end of cycle 3
  + This value can be passed to C stage (ALU) of I2 in cycle 4
  + Introduce MUX A before operand A input of ALU
    - Can select from RA or forwarded value from RZ

Clock 1 2 3 4 5 6

I1 F D C M W

I2 F D C M W

* Memory stalls – caused by delays in memory responses
  + i.e. if cache miss, accessing main memory ⇒ subsequent instructions are stalled

Clock 1 2 3 4 5 6 7

I1 F D C M W

I2 F D C M …

I3 F D C …

* Load-use stalls – RAW dependency even if cache hit + operand forwarding
  + E.g. consider:

LDR R2, (R3) ; result available in M stage (cycle 4)

SUB R9, R2, #30 ; R2 used in C stage (cycle 4)

* + I2 requires 1 cycle of stall

Clock 1 2 3 4 5 6 7

I1 F D C M W

I2 F D C M W

* Unconditional branch delays
  + Branch instructions alter sequential execution – can’t always just fetch I2 while I1 is being decoded
  + Consider an unconditional branch instruction I1 with target Ix
  + Target Ix is known for I1 in C stage (cycle 3), but I2 & I3 are fetched in cycles 2 & 3
  + Instructions fetched down the wrong path need to be squashed – results in branch penalty
    - Here I2 & I3 are squashed – branch penalty = 2 cycles
  + Can introduce an adder in D stage for computing branch address
    - Reduce branch penalty down to 1 cycle

Clock 1 2 3 4 5 6

I1 F D C M W

I2 F (squashed)

Ix F D C …

* Conditional branch delays
  + Introduce comparator in D stage for conditional branches
  + Branch penalty = 1
* Delayed branching
  + Branch delay slot – location immediately following branch
  + Instruction in delay slot is always fetched
  + Instead of conditionally discarding delay slot instruction, always let it complete execution
  + Find an instruction before the branch and reorder so that it’s in the delay slot (if data dependencies permit)
  + The instruction is useful & is always executed regardless of branch decision – branch penalty = 0
  + If not possible, inserting NOP in delay slot (always delay one cycle)
  + Ex:

ADD R7, R8, R9

B\_if\_[R3]=0 TARGET

I2 …

* + - I2 will be squashed; branch penalty = 1
    - No dependency exists between the ADD and B; reorder ADD into delay slot
  + Improved:

B\_if\_[R3]=0 TARGET

ADD R7, R8, R9

* + - I2 …
    - ADD is always performed, no instructions squashed; branch penalty = 0
* Static branch prediction
  + Assume branch is not taken
  + Branches at end of loops are usually taken, i.e. branch offset < 0
  + Branches at start of loops are usually not taken, i.e. branch offset > 0
  + Otherwise, assume randomly (accuracy ~50%)
* Dynamic branch prediction
  + Keep track of branching decisions to predict future branching
  + 2-state algorithm:
    - Assume branching decision will be same as last time
    - LT – likely to be taken
    - LNT – likely not to be taken
    - LNT → LT if branch is taken (BT), otherwise remains LNT
    - LT → LNT if branch is not taken (BNT), otherwise remains LT
    - Requires only one bit to keep track of
    - First & last passes in loops will be incorrectly predicted
  + 4-state algorithm:
    - ST – strongly likely to be taken
    - SNT – strongly likely not to be taken
    - ST → LT if BNT; no change if BT
    - LT → SNT if BNT; LT → ST if BT
    - LNT → ST if BT; LNT → SNT if BNT
    - SNT → LNT if BT; no change if BNT
    - Requires 2 bits
    - Repeated execution of loops will only have 1 misprediction
* Branch target buffer (BTB)
  + Allows branch prediction to be done in F stage (cycle 1)
  + Improves likelihood of fetching the correct instruction in cycle 2 − reduce penalty
  + BTB is a lookup table mapping addresses of branch instructions → their prediction state bits & target addresses (from previous occurrences)
  + With predictor and BTB in F stage:
    - Cycle 1 – lookup branch instruction address in BTB to get prediction & target address
    - Cycle 2 – fetch next instruction using prediction
      * Compare to check if prediction is correct (D stage)
      * If mispredicted, squash & fetch correct instruction in cycle 3
* **Performance**
* Execution time: non-pipelined
  + T – execution time (s/program)
  + N – dynamic instruction count (instructions/program)
  + S – clock cycles to fetch & execute an instruction (cycles/instruction)
  + R – clock rate/CPI (cycles/s)
  + **T = N \* S / R**
* Throughput (P)
  + **P = R/S** (instructions/s)
  + For non-pipelined (assuming no cache misses), S = 5
  + For pipelined (assuming no stalls), S = 1
    - **S = 1 + δstall**
    - E.g. for program P, 25% of instructions are loads & 40% of loads incur load-use stalls of 1 cycle
      * δstall = δload = 0.25 \* 0.40 \* 1 = 0.10
      * Pp = R/(1 + 0.10) = 0.91R
* **Superscalar organization**
  + Scalar pipelines – allow max of 1 instruction/cycle
  + Superscalar – multiple instructions are fetched & executed at the same time
  + Components required:
    - Fetch unit − inserts ≥ 2 instructions into instruction queue
    - Dispatch unit − takes ≥ 2 instructions from instruction queue, decodes them & sends them execution units
    - Multiple execution units − enable multiple instruction issue; results sent to completion unit
      * Minimally contains load/store unit & arithmetic unit
    - Completion unit − writes multiple results to registers
      * With 2 execution units, require 4 read ports & 2 write ports
      * Must avoid multiple writes to the same register at the same time
* Branching & data dependencies in superscalar processors
  + Speculative execution
    - Instructions based on prediction are speculatively executed, but results are not written until prediction is confirmed
  + Reservation stations – buffers in execution units
    - A dependent instruction must wait for its operands to become available
    - Reservation stations hold operands for instructions awaiting execution
  + Out-of-order execution
    - If two instructions have no dependencies, they no ordering constraints
    - OOO execution leads to imprecise exceptions possibly being raised
  + Execution completion
    - Instructions can be allowed to execute out-of-order, but their results must be committed in strict order (commitment unit is responsible for this)
    - Register renaming
      * Use temporary registers to hold results of instructions before they are committed
    - Reorder buffer – queue in commitment unit
      * Dispatched instructions are entered in order
      * Only write result to dest registers when instruction reaches the head of queue
    - Dispatch of instructions only proceeds if all needed resources are available
      * E.g. temporary registers, reservation station, reorder buffer
      * Deadlock condition – two instructions need each other’s resources