Superscalar Processors

Superscalar Processor

- **Multiple Independent Instruction Pipelines; each with multiple stages**
- **Instruction-Level Parallelism**
- **determine dependencies between nearby instructions**
  - **input of one instruction depends upon the output of a preceding instruction**
- **locate nearby independent instructions**
  - **issue & complete instructions in an order different than specified in the code stream**
- **uses branch prediction methods rather than delayed branches**
- **RISC or CICS**

Super Pipelined Processor

- **Pipeline stages can be segmented into n distinct non-overlapping parts each of which can execute in $^1/_n$ of a clock cycle**

Limitations of Instruction-Level Parallelism

- **True Data Dependency (Flow Dependency, Write-After-Read [WAR] Dependency)**
  - **Second Instruction requires data produced by First Instruction**
- **Procedural Dependency**
  - **Branch Instruction – Instructions following either**
    - **Branches-Taken**
    - **Branches-NotTaken**

    } **have a procedural-dependency on the branch**
  - **Variable-Length Instructions**
    - **Partial Decoding is required prior to the fetching of the subsequent instruction, i.e., computing PC value**
- **Resource Conflicts**
  - **Memory, cache, buses, register ports, file ports, functional units access**
- **Output Dependency**
  - **See "In-Order Issue → Out-of-Order Completion" below**
- **Anti-dependency**
  - **See "Out-of-Order Issue → Out-of-Order Completion" below**

Instruction-Level Parallelism

- **Instructions in sequence are independent**
  → instructions can be executed in parallel by overlapping

- **Degree of Instruction-Level Parallelism depends upon**
  - **Frequencies of True Data Dependencies & Procedural Dependencies in the code which is dependent upon the Instruction Set Architecture & the Application**
  - **Operational Latency – the time until the result of an instruction is available for use in a subsequent instruction, i.e., Execution Completion Time**

Machine Parallelism

- **Number of instructions that can be fetched and executed simultaneously, i.e., number of parallel pipelines**
- **Sophistication, i.e., speed, of the mechanisms that the processor uses to locate independent instructions**

Instruction Issue Policy

- **Instruction Issue**
  - **process of initiating instruction execution in the processor's functional units**
  - **occurs when instruction moves from the decode stage to the first execute stage of the pipeline**

- **Instruction Issue Policy**
  - **protocol used to issue instructions**
  - **looking ahead of the current point of execution to locate instructions that can be placed into the pipeline & executed**

- **Types of Orderings**
  - **Order in which instructions are fetched**
  - **Order in which instructions are executed**
  - **Order in which instructions update register contents & memory locations**

- **Instruction Issue Policy Categories**
  - **In-Order Issue ➔ In-Order Completion**
    - **superscalar pipeline specifications (example)**
      - **fetch/decode two instructions per time period**
      - **digital arithmetic functional units (2)**
      - **floating-point arithmetic functional unit (1)**
      - **write-back pipeline stages (2)**
    - **code fragment (six instructions) constraints**
      - **I1 requires two cycles to complete**
      - **I3 & I4 conflict for the same functional unit**
      - **I5 depends upon a value produced by I4**
      - **I5 & I6 conflict for a functional unit**
    - **Procedure**
      - **fetch next two instructions into the decode stage**
      - **issuing of instructions must wait until current instructions have passed the decode pipeline stages**
      - **conflict for a functional unit ➔ issuing of instructions must temporary halt**
      - **functional unit requires more than one cycle to generate a result ➔ issuing of instructions must temporary halt**

(figure 14.4 page 531)

- **In-Order Issue ➔ Out-of-Order Completion**
  - **Procedure**
    - **fetch next two instructions into the decode stage**
    - **number of instructions in execute stages ≤ maximum degree of machine parallelism across all functional units**
    - **instruction issuing stalled by**
      - **resource conflict**
      - **data dependency**
      - **procedural dependency**
    - **output dependency, i.e., write-after-write (WAW) dependency**
    - **code fragment (four instructions)**
      - **I1: R3 ← R3 op R5**
      - **I2: R4 ← R3 + 1**
      - **I3: R3 ← R5 + 1**
      - **I4: R7 ← R3 op R4**
    - **Constraints**
      - **I1 must execute before I2** **I1 produces R3 contents required by I2**
      - **I3 must execute before I4** **I3 produces R3 contents required by I4 (RAW)**
      - **I3 must complete after I1** **I3 must write R3 after I1 writes R3 (WAW)**
    - **conflict for a functional unit ➔ issuing of instructions must temporary halt**
    - **functional unit requires more than one cycle to generate a result ➔ issuing of instructions must temporary halt**
    - **issuing an instruction must stall if its result might later be overridden by an instruction issued earlier which takes longer to complete (WAW)**

Instruction Issue Logic

- **more complex**
- **interrupts & exceptions handling**
  - **resumption – some instructions which logically follow the interrupted instruction may have already completed and perhaps must not be executed again**

- **Out-of-Order Issue ➔ Out-of-Order Completion**
  - **In-Order Issue – decode instructions up to the point of dependency; cannot look ahead of dependency to subsequent instructions independent of those in the pipeline that could be usefully introduced into the pipeline**

  - **Out-of-Order Issue**
    - **decouple the decode stages from the execute stages of the pipeline**
    - **Instruction Window Buffer**
      - **decoded instructions are placed in the Instruction Window Buffer**
      - **Instruction Window Buffer full ➔ instruction fetching & decoding must temporarily halt**
      - **functional unit becomes available ➔ instruction from the Instruction Window Buffer may be issued provided**
        - **it needs the particular functional unit available**
        - **no conflicts or dependencies block the instruction**
    - **processor has lookahead capability allowing it to identify independent instructions that can be brought into the execute stage**

  - **Procedure**
    - **fetch next two instructions into the decode stage**
    - **during each cycle, subject to the buffer size, two instructions move from the decode stage to the Instruction Window Buffer**
    - **($\forall$ instruction) $\in$ Instruction Window Buffer ➔ processor has sufficient information to decide when it can be issued**
    - **more instructions are available for issuing ➔ reducing probability of pipeline stage stall**
    - **number of instructions in execute stages $\leq$ maximum degree of machine parallelism across all functional units**
    - **instruction issuing stalled by**
      - **resource conflict**
      - **data dependency**
      - **procedural dependency**
      - **antidependency, read-after-write dependency (RAW), i.e., second instruction destroys a value that is required by the first instruction**
        - **code fragment (four instructions)**

          I1: R3 ← R3 op R5
          I2: R4 ← R3 + 1
          I3: R3 ← R5 + 1
          I4: R7 ← R3 op R4
        - **I3 cannot complete execution before I2 begins execution and has fetched its operands**
        - **I3 updates R3 which is a source register for I2 (RAW)**

  - **Reorder Buffer & Tomasulo's Algorithm – Appendix I**

- **Register Renaming**
  - **execution sequence & data flow ➔ RAW data dependencies & resource conflicts**
  - **out-of-order instruction issuing or completion ➔ WAW & WAR dependencies**
    - **values in registers no longer reflect sequence of values dictated by the program flow**
    - **antidependencies & output dependencies ➔ storage conflicts, i.e., multiple instructions are completing for the same registers ➔ pipeline constraints**

      compiler register optimization techniques ➔ maximize register conflicts

  - **registers are allocated dynamically by the processor hardware; these registers are associated with the values needed by instructions at certain points in time**
    - **instruction with a register as a destination operand executes ➔ new register is allocated for that value**
    - **subsequent instructions that access that value as a source operand in that register must undergo a renaming process, i.e., the register references must be revised to refer to the registers containing the needed values**
    - **the same original register references in several different instructions may refer to different actual registers if different values are indicated**
    - **code fragment (four instructions)**

      I1: $R3_b \leftarrow R3_a$ op $R5_a$
      I2: $R4_b \leftarrow R3_b + 1$
      I3: $R3_c \leftarrow R5_a + 1$
      I4: $R7_b \leftarrow R3_c$ op $R4_b$
    - **register allocation protocol**
      - **Rx_a is allocated before Rx_b**
      - **Rx_b is allocated before Rx_c**

      allocation made for a particular logical register ➔ subsequent instruction references to that logical register as a source operand are made to refer to the most recently allocated hardware register in the program sequence of instructions

Creation of $R3_c$ in I3 avoids
→ RAW dependency in I2
→ OUTPUT dependency on I1
→ Does not interfere w2ith the correct value being accessed by I4
hence with renaming I3 can be issued immediately

Without register renaming, I3 cannot be issued until I1 is complete and I2 is issued

- **Scoreboarding – Appendix I**

Machine Parallelism – page 535 figure 14.5

Branch Prediction

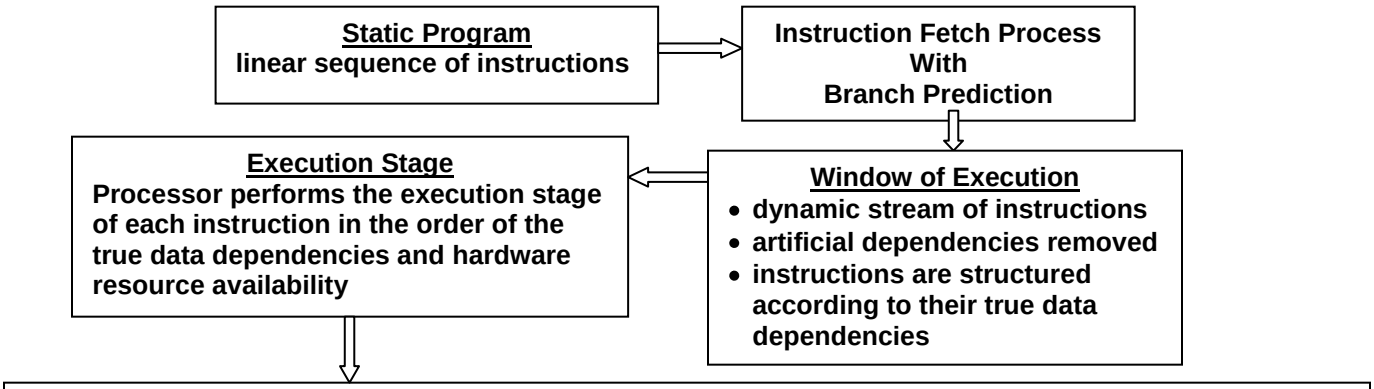
- **Intel 80486 fetched both**
  - **next sequential instruction**
  - **speculatively fetched the branch target instruction**
  - **two pipeline stages between prefetch & execution**

  two cycle delay when branch is taken

- **RISC machines**
  - **delayed branch strategy**
  - **calculate the single instruction that follows the branch instruction**
  - **if necessary, fetch a new instruction stream**

- **superscalar machines**
  - **static branch prediction**
    - **always predict branch not taken**
    - **always predict branch taken**
  - **dynamic branch prediction based on branch history analysis**
    - **branch based on recent history**
    - **branch based on long term history**
    - **branch based on a weighted historical computation**

Superscalar Execution



Committing or Retiring the Instruction

Instructions are conceptually placed back into the original linear sequence and the results are recorded

- **parallel, multiple pipelines ➔ instructions complete in an order different from the static program**
- **branch prediction & speculative execution ➔ instructions may complete execution & be abandoned**
- **permanent storage & program-visible registers cannot be updated immediately**
- **results must be held in some sort of temporary storage until it is determined that the sequential model would have executed the instruction**

Superscalar Implementation (Hardware)

- **Instruction Fetch Strategies**
  - **Simultaneously Fetching Multiple Instructions**
  - **Predicting the Outcomes of, and Fetching Beyond, Conditional Branch Instructions**
  - **Requires the use of Multiple Pipeline Fetch & Decode Stages**
  - **Requires the use of Branch Prediction Logic**
- **True Dependencies Logic**
  - **Logical Register Assignments**
  - **Tracking the Register Assignments to where they are needed during execution**
- **Mechanisms for Issuing Multiple Instructions in Parallel**
- **Resources for Parallel Execution of Multiple Instructions**
  - **Multiple Functional Units**
  - **Memory Hierarchies capable of simultaneously servicing Multiple Memory References**
- **Mechanisms for Committing the Process State in the Original Sequential Order**

Intel 80xxx

80386 traditional CICS non-pipelined machine

80486 pipelined processor
- **reduced integer operations average latency from two to four cycles to one cycle**
- **executed a single instruction per cycle**

Pentium implemented a limited superscalar system – two separate integer execution units

Pentium Pro implemented a full featured superscalar system

Pentium 4
- **Operational Protocol**
  - **fetch instructions from memory in static program order**
  - **translate each instruction into one or more micro-operations**
  - **execute the micro-ops in a superscalar pipeline organization, i.e., micro-ops can be executed out of order**
  - **results of each micro-op execution are submitted to the register set in the order of the original program flow**

- **Architecture**
  - **outer CICS shell**
    - **Front End**
      - **Micro-Op Generation**

page 538-539
figures 14.7 & 14.8

        - **Branch Target Buffer (BTB)**
          - **branch prediction**
          - **static branch prediction determines the next effective program counter value**
        - **Instruction Translation Lookaside Buffer (I-TLB)**
          - **Translates the linear instruction pointer address into a physical address required to access the L2 cache**
        - **L2 Cache is feed sequential static code from the original program**
          - **each cache line fetch in L2 contains the next sequential instruction**
        - **L1 Instruction Trace Cache**
          - **contains micro-op code**
        - **Fetch/Decode Unit, with the aid of the BTB & I-TLB,**
          - **retrieves instructions, using in-order issue, unless it is altered by branch prediction by the BTB & I-TLB, from the L2 cache (static code) in 64 byte packets and feeds the L1 instruction cache, i.e., Trace Cache**
          - **scans the bytes to determine the instruction boundaries**
          - **translates each instruction into one to four micro-ops**
            - **each micro-op is a 118 bit RISC instruction**
            - **micro-op (RISC) are stored in the L1 Trace Cache**
        - **processor operates from the Trace Cache**
        - **cache miss → In-Order Front End feeds new instructions to the Trace Cache**

- **inner RISC core – pipeline 20 stages – selected instructions use a longer pipeline**
  - **Trace Cache Next Instruction Pointer**
    - **BTB**
      - **caches the history of recent executions of branch instructions**
      - **dynamic branch prediction strategy based on the recent history**
      - **if a particular branch instruction is in the BTB then its branch history is used to predict the current branch strategy (dynamic branch prediction)**
      - **after the branch execution, the BTB entry is updated with the recent results**
      - **if a particular branch instruction is not in the BTB then**
        - **an entry is made; an older entry is deleted if necessary**
        - **static branch prediction algorithm**
          - **IP-relative backward conditional branches (loops) predict that branch is taken**
          - **IP-relative forward conditional branches predict that branch is not taken**
          - **Branch addresses that are not IP-relative, predict**
            - ✓ **taken if branch is a return**
            - ✓ **predict not taken otherwise**
      - **four-way set-associative cache with 512 lines**
        - **tag is the branch address**
        - **branch destination address for last time branch was taken**
        - **4 bit history field to indicate the last 16 times with the specified branch**
        - **Yeh's Algorithm used for dynamic branch prediction**
  - **Trace Cache Fetch**
    - **Trace Cache assembles the decoded micro-ops into program-ordered sequences, i.e., traces**
    - **Micro-ops are fetched sequentially from the Trace Cache subject to branch prediction logic**
    - **Instructions requiring more than four micro-ops are transferred to the microcode ROM unit which is a microprogrammed control unit; after the ROM finishes the sequencing of the current micro-ops, fetching resumes from the Trace Cache**
  - **Drive**
    - **pipeline delivers micro-ops from the Trace Cache to the Rename/Allocator module**
  - **Allocate Stage (allocates resources)**
    - **resource required by micro-op arriving during a clock cycle is unavailable → allocator stalls the pipeline**
      - **three micro-ops arrive at Allocator during each clock cycle**
    - **allocates a reorder buffer (ROB) entry which tracks the completion status of a selected micro-op which is currently in process**
      - **126 micro-ops could be in process at any time**
    - **allocates an integer or floating-point register entry for the result data**
    - **possibly allocates a load or store buffer used to track one of the 48 loads or 24 stores in the pipeline**
    - **allocates an entry in one of the two micro-op queues in front of the instruction schedulers**

Reorder Buffer (ROB)

- **circular buffer**
- **holds a maximum of 126 micro-ops**
- **contains 128 hardware registers**
- **buffer entry fields**
  - **State**
    - ♦ **Scheduled for Execution**
    - ♦ **Dispatched for Execution**
    - ♦ **Completed Execution**
    - ♦ **Ready for Retirement**
  - **Memory Address**
    - ♦ **address of instruction that generated the micro-op**
  - **Micro-op**
  - **Alias Register**
    - ♦ **If the micro-op references one of the 16 architectural registers, then the entry redirects tat reference to one of the 128 hardware registers**

---

micro-ops enter ROB in-order

micro-ops are dispatched from the ROB to the Dispatch/Execute unit out-of-order

dispatch criterion

- **appropriate execution unit is available**
- **all necessary data items required by micro-code are available**

micro-ops are retired from the ROB in-order

- **micro-ops are retired oldest first after each micro-op are designated as ready for retirement**

♦ **Register Renaming**
- — **Architectural Registers (16) available to programmers**
  - ▪ **Floating point registers (8)**
  - ▪ **EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP**
- — **Physical Registers (128) available to hardware**
- — **renaming : Architectural Registers → Physical Registers**
  - ▪ **removes false dependencies caused by a limited number of Architectural Registers**
  - ▪ **preserves true dependencies (RAW)**

♦ **Micro-Op Queuing page 540, figure 14.9f**
- — **After Resource Allocation & Register Renaming micro-ops are placed in one of two micro-op queues**
- — **Micro-ops Queue for micro-ops requiring memory operations**
- — **Micro-ops Queue for micro-ops that do not require memory operations**
- — **each queue, individually, obeys FIFO access**
- — **between queues there is no access discipline**

- **Micro-Op Scheduling and Dispatching**
  - **Schedulers look for micro-ops with Status indicating that they have all operands**
    - **Execution unit is available for the indicated micro-op ➔ micro-op is dispatched**
    - **More than one micro-op is available for an available execution unit, micro-ops are dispatched in FIFO order**
    - **Maximum of 6 micro-ops can be dispatched in a single cycle**
  - **Ports attach Schedulers to Execution Units (4)**
    - **Port 1 handles Simple Integer Operations & Branch Mispredictions**
    - **Port 0 handles other Integer & Floating Point Instructions**
    - **MMX Execution Units are allocated between Port 0 and Port 1**
    - **Ports 2 and Port 3 are used for memory loads and Stores**

- **Integer & Floating Point Execution Units**

  Integer Register Files
  Floating Point Register Files } **source for pending operations by execution units**

  - **Execution Units retrieve information from the Register Files and the L1 Data Cache**
  - **Separate pipeline stage is used to compute Flags which are input to branch instructions**
  - **Subsequent pipeline stage compares the actual branch result with the prediction**
    - **If prediction is wrong, there are micro-ops in various stages of the pipeline that must be removed; then the proper branch destination is provided to the Branch Predictor during a Drive Stage, thus restarting the pipeline from a new target address**