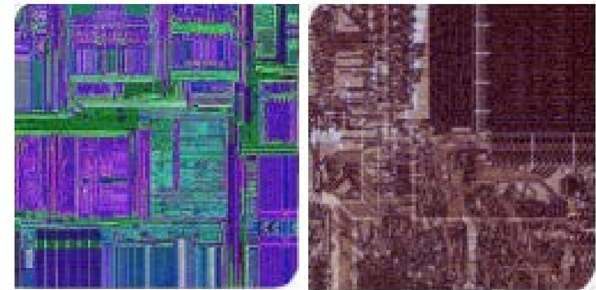
# Hardware-based Speculation



# Hardware-based Speculation

- To exploit instruction-level parallelism, maintaining control dependences becomes an increasing burden.
- For a processor executing multiple instructions per clock, just predicting branches accurately may not be sufficient to generate the desired amount of instruction-level parallelism
- Overcoming control dependence is done by speculating on the outcome of branches and executing the program as if the guesses were correct.
- Speculation ⇒ fetch, issue, and execute instructions as if branch predictions were always correct
- Dynamic scheduling ⇒ only fetches and issues such instructions
- Mechanisms are required to handle the situation when the speculation goes incorrect.

# Hardware-based Speculation

- Combines three key ideas
  - dynamic branch prediction to choose which instructions to execute
  - speculation to allow the execution of instructions before the control dependences are resolved
    - ability to undo the effects of an incorrectly speculated sequence
  - dynamic scheduling to deal with the scheduling of different combinations of basic blocks

- Data flow execution - Predicted flow of data values to choose when to execute instructions. Operations execute as soon as their operands are available

# Hardware-based Speculation

- The key idea
  - allow instructions to execute out of order
  - force instructions to commit in order
  - prevent any irrevocable action (such as updating state or taking an exception) until an instruction commits.
- Hence:
  - Must separate execution from allowing instruction to finish or "commit"
    - instructions may finish execution considerably before they are ready to commit.
- This additional step called instruction commit
- Adding this commit phase to the instruction execution sequence requires an additional set of hardware buffers that hold the results of instructions that have finished execution but have not committed – Reorder buffer.

# Reorder Buffer (ROB)

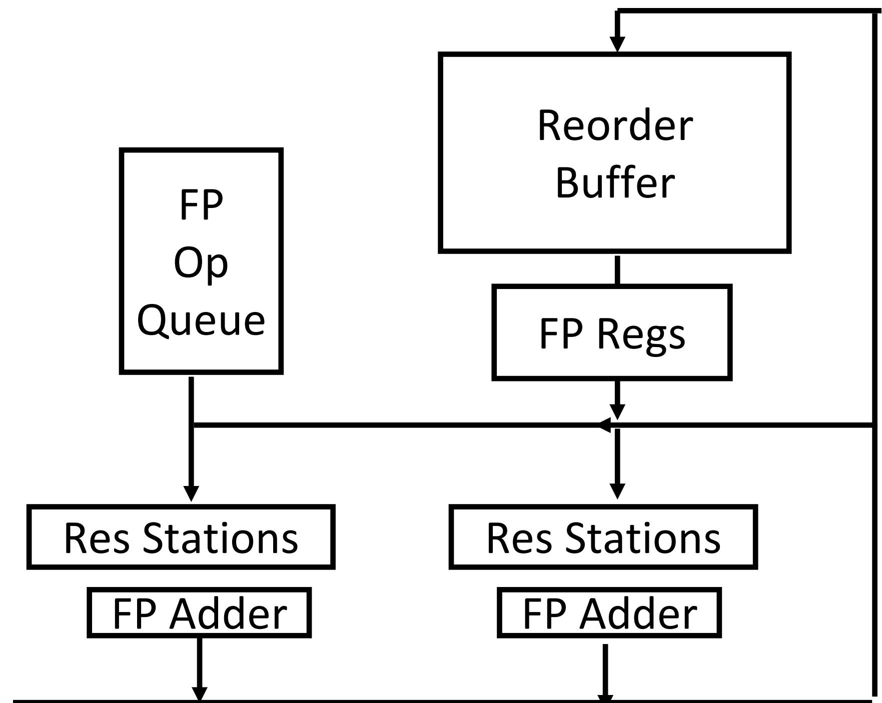
- Holds the result of an instruction between the time the operation associated with the instruction completes and the time the instruction commits
- Source of operands for instructions, just as the reservation stations provide operands in Tomasulo's algorithm
- In Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file
- With speculation, the register file is not updated until the instruction commits

# Reorder Buffer Structure

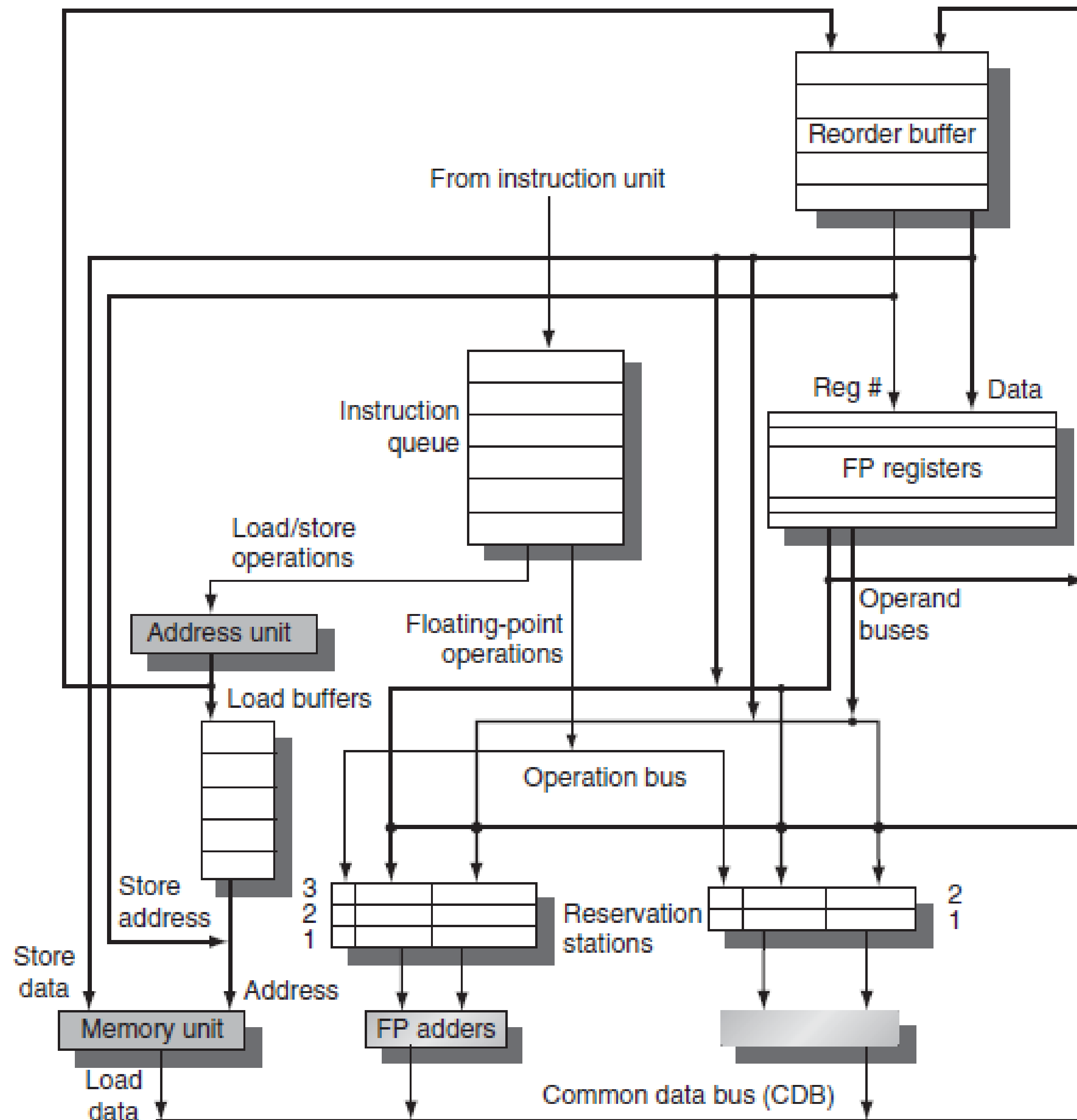
- Four fields - instruction type, destination field, value field, and ready field
- Instruction type field indicates whether the instruction is a branch (and has no destination result), a store (which has a memory address destination), or a register operation (ALU operation or load, which has register destinations)
- Destination field supplies the register number (for loads and ALU operations) or the memory address (for stores) where the instruction result should be written
- Value field is used to hold the value of the instruction result until the instruction commits
- Ready field indicates that the instruction has completed execution, and the value is ready

# Reorder Buffer Operation

- Addition of the ROB and the elimination of the store buffer, whose function is integrated into the ROB
- Renaming function of the reservation stations is replaced by the ROB
- Tag a result using the ROB entry number rather than using the reservation station number



# Reorder Buffer Operation



# Reorder Buffer Operation

- *Issue*
  - Get an instruction from the instruction queue
  - Issue the instruction if there is an empty reservation station and an empty slot in the ROB
  - Send the operands to the reservation station if they are available in either the registers or the ROB
  - Update the control entries to indicate the buffers are in use
  - The number of the ROB entry allocated for the result is also sent to the reservation station, so that the number can be used to tag the result when it is placed on the CDB.
  - If either all reservations are full or the ROB is full, then instruction issue is stalled until both have available entries

# Reorder Buffer Operation

- *Execute*
  - If one or more of the operands is not yet available, monitor the CDB while waiting for the register to be computed - checking for RAW hazards.
  - When both operands are available at a reservation station, execute the operation.
  - Instructions may take multiple clock cycles in this stage, and loads still require two steps in this stage.
  - Execution for a store is only effective address calculation

# Reorder Buffer Operation

- *Write Result*
  - When the result is available, write it on the CDB (with the ROB tag sent when the instruction issued) and from the CDB into the ROB, as well as to any reservation stations waiting for this result
  - Mark the reservation station as available.
  - If the value to be stored is available, it is written into the Value field of the ROB entry for the store.
  - If the value to be stored is not available yet, the CDB must be monitored until that value is broadcast, at which time the Value field of the ROB entry of the store is updated.

# Reorder Buffer Operation

- *Commit*
  - Final stage of completing an instruction, after which only its result remains – completion or graduation
  - Normal commit - instruction reaches the head of the ROB and its result is available in the buffer; updates the register with the result and removes the instruction from the ROB
  - Committing a store - similar to normal commit except that memory address is updated rather than a result register
  - Branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation is incorrect. The ROB is flushed and execution is restarted at the correct successor of the branch. If the branch was correctly predicted, the branch is finished.

# Example

- Assume the latencies for the floating-point functional units as: add is 2 clock cycles, multiply is 6 clock cycles, and divide is 12 clock cycles. Using the code segment below, show what the status tables look like when the MUL.D is ready to go to commit.

```
L.D    F6,32(R2)
L.D    F2,44(R3)
MUL.D  F0,F2,F4
SUB.D  F8,F2,F6
DIV.D  F10,F0,F6
ADD.D  F6,F8,F2
```

# Example

**Reorder buffer**

| Entry | Busy | Instruction | | State | Destination | Value |
|---|---|---|---|---|---|---|
| 1 | No | `L.D` | `F6,32(R2)` | Commit | F6 | `Mem[32 + Regs[R2]]` |
| 2 | No | `L.D` | `F2,44(R3)` | Commit | F2 | `Mem[44 + Regs[R3]]` |
| 3 | Yes | `MUL.D` | `F0,F2,F4` | Write result | F0 | #2 × `Regs[F4]` |
| 4 | Yes | `SUB.D` | `F8,F2,F6` | Write result | F8 | #2 – #1 |
| 5 | Yes | `DIV.D` | `F10,F0,F6` | Execute | F10 | |
| 6 | Yes | `ADD.D` | `F6,F8,F2` | Write result | F6 | #4 + #2 |

**Reservation stations**

| Name | Busy | Op | Vj | Vk | Qj | Qk | Dest | A |
|---|---|---|---|---|---|---|---|---|
| Load1 | No | | | | | | | |
| Load2 | No | | | | | | | |
| Add1 | No | | | | | | | |
| Add2 | No | | | | | | | |
| Add3 | No | | | | | | | |
| Mult1 | No | `MUL.D` | `Mem[44 + Regs[R3]]` | `Regs[F4]` | | | #3 | |
| Mult2 | Yes | `DIV.D` | | `Mem[32 + Regs[R2]]` | #3 | | #5 | |

**FP register status**

| Field | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Reorder # | 3 | | | | | | 6 | | 4 | 5 |
| Busy | Yes | No | No | No | No | No | Yes | . . . | Yes | Yes |

# Example

- Show what the status table look like when the loop is iterated twice. Assume that the L.D and MUL.D from the first iteration have committed and all other instructions have completed execution.

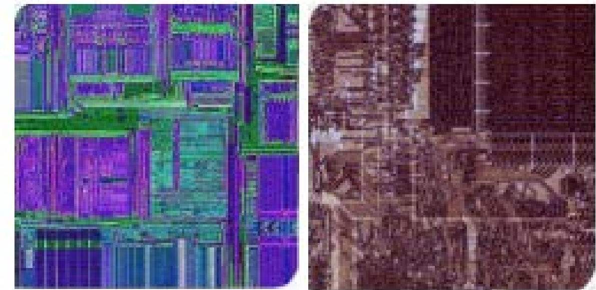
```
Loop: L.D    F0,0(R1)
      MUL.D  F4,F0,F2
      S.D    F4,0(R1)
      DADDIU R1,R1,-8
      BNE    R1,R2,Loop;  branches if R1¦R2
```

# Example

| Reorder buffer | | | | | |
|---|---|---|---|---|---|
| **Entry** | **Busy** | **Instruction** | **State** | **Destination** | **Value** |
| 1 | No | L.D F0,0(R1) | Commit | F0 | Mem[0 + Regs[R1]] |
| 2 | No | MUL.D F4,F0,F2 | Commit | F4 | #1 × Regs[F2] |
| 3 | Yes | S.D F4,0(R1) | Write result | 0 + Regs[R1] | #2 |
| 4 | Yes | DADDIU R1,R1,#-8 | Write result | R1 | Regs[R1] – 8 |
| 5 | Yes | BNE R1,R2,Loop | Write result | | |
| 6 | Yes | L.D F0,0(R1) | Write result | F0 | Mem[#4] |
| 7 | Yes | MUL.D F4,F0,F2 | Write result | F4 | #6 × Regs[F2] |
| 8 | Yes | S.D F4,0(R1) | Write result | 0 + #4 | #7 |
| 9 | Yes | DADDIU R1,R1,#-8 | Write result | R1 | #4 – 8 |
| 10 | Yes | BNE R1,R2,Loop | Write result | | |

| FP register status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Field** | **F0** | **F1** | **F2** | **F3** | **F4** | **F5** | **F6** | **F7** | **F8** |
| Reorder # | 6 | | | | 7 | | | | |
| Busy | Yes | No | No | No | Yes | No | No | ... | No |

# Exploiting ILP Using Multiple Issue & Static Scheduling



# Exploiting ILP Using Multiple Issue and Static Scheduling

- The goal of multiple-issue processors, is to allow multiple instructions to issue in a clock cycle.
- Multiple-issue processors come in three major flavors:
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - Dynamically scheduled superscalar processors
- The two types of superscalar processors issue varying numbers of instructions per clock and use in-order execution if they are statically scheduled or out-of order execution if they are dynamically scheduled.
- VLIW processors issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction.
- VLIW processors are inherently statically scheduled by the compiler.

# Exploiting ILP Using Multiple Issue and Static Scheduling

| Common name | Issue structure | Hazard detection | Scheduling | Distinguishing characteristic | Examples |
|---|---|---|---|---|---|
| Superscalar (static) | Dynamic | Hardware | Static | In-order execution | Mostly in the embedded space: MIPS and ARM, including the ARM Cortex-A8 |
| Superscalar (dynamic) | Dynamic | Hardware | Dynamic | Some out-of-order execution, but no speculation | None at the present |
| Superscalar (speculative) | Dynamic | Hardware | Dynamic with speculation | Out-of-order execution with speculation | Intel Core i3, i5, i7; AMD Phenom; IBM Power 7 |
| VLIW/LIW | Static | Primarily software | Static | All hazards determined and indicated by compiler (often implicitly) | Most examples are in signal processing, such as the TI C6x |
| EPIC | Primarily static | Primarily software | Mostly static | All hazards determined and indicated explicitly by the compiler | Itanium |

# VLIW Approach

- Use multiple, independent functional units
- Rather than attempting to issue multiple, independent instructions to the units, a VLIW packages the multiple operations into one very long instruction
- Multiple operations are placed in one instruction with maximum issue rate
- Consider a VLIW processor with instructions that contain five operations, including one integer operation or branch, two floating-point operations, and two memory references - instruction have a set of fields for each functional unit
- Intel Itanium 1 and 2 contain six operations per instruction packet

# Example

- Suppose a VLIW could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle. Show an unrolled version of the loop

$$x[i] = x[i] + s$$

for such a processor. Unroll as many times as necessary to eliminate any stalls. Ignore delayed branches.

```
Loop: L.D F0,0(R1)          ;F0=array element
      ADD.D F4,F0,F2        ;add scalar in F2
      S.D F4,0(R1)          ;store result
      DADDUI R1,R1,#-8      ;decrement pointer 8 bytes
      BNE R1,R2,Loop        ;branch R1!=R2
```

# Example

| Memory reference 1 | Memory reference 2 | FP operation 1 | FP operation 2 | Integer operation/branch |
|---|---|---|---|---|
| L.D F0,0(R1) | L.D F6,-8(R1) | | | |
| L.D F10,-16(R1) | L.D F14,-24(R1) | | | |
| L.D F18,-32(R1) | L.D F22,-40(R1) | ADD.D F4,F0,F2 | ADD.D F8,F6,F2 | |
| L.D F26,-48(R1) | | ADD.D F12,F10,F2 | ADD.D F16,F14,F2 | |
| | | ADD.D F20,F18,F2 | ADD.D F24,F22,F2 | |
| S.D F4,0(R1) | S.D F8,-8(R1) | ADD.D F28,F26,F2 | | |
| S.D F12,-16(R1) | S.D F16,-24(R1) | | | DADDUI R1,R1,#-56 |
| S.D F20,24(R1) | S.D F24,16(R1) | | | |
| S.D F28,8(R1) | | | | BNE R1,R2,Loop |

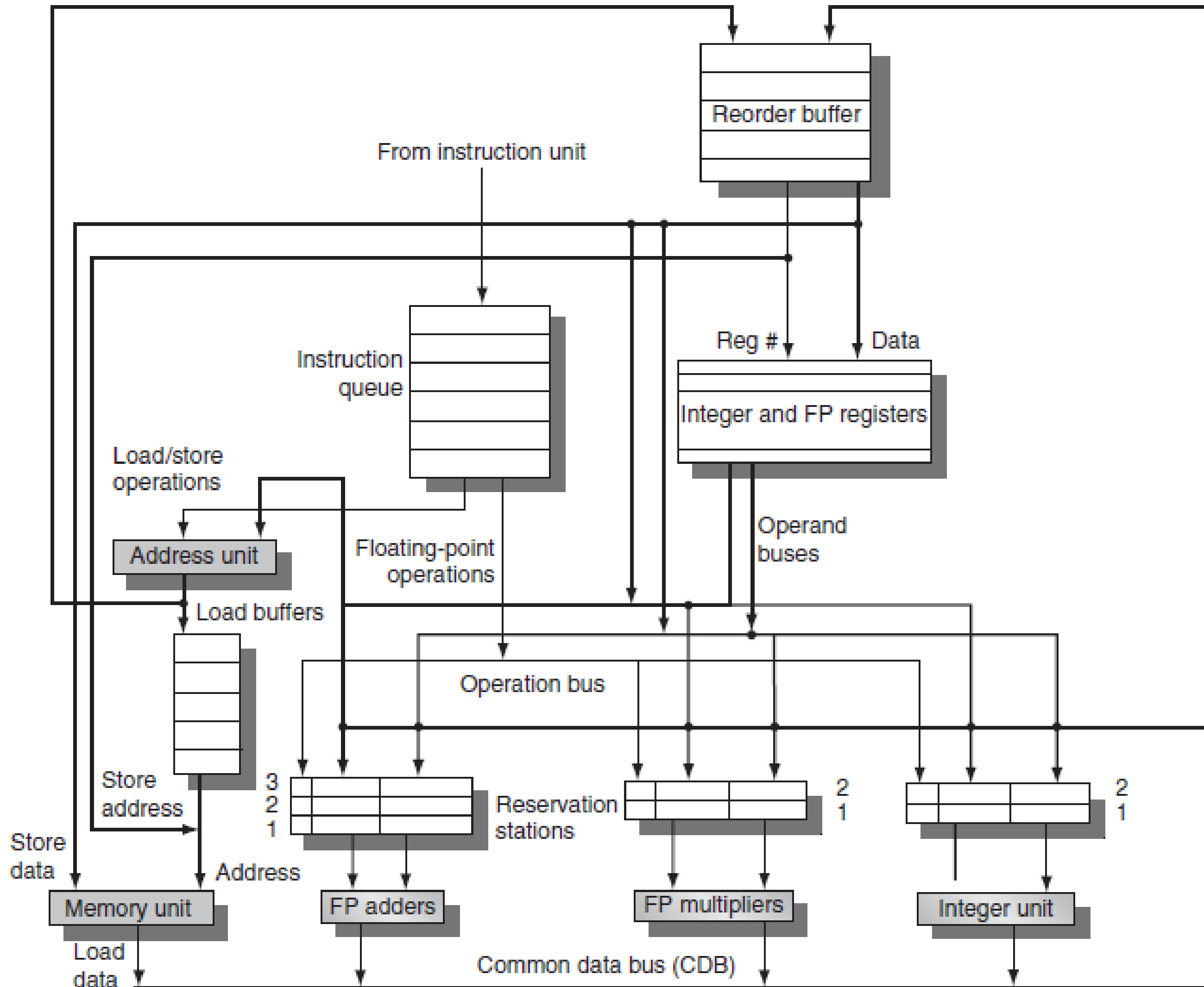
- Loop has been unrolled to make seven copies of the body, which eliminates all stalls and runs in 9 cycles.
- Yields a running rate of seven results in 9 cycles, or 1.29 cycles per result
- The issue rate is 23 operations in 9 clock cycles, or 2.5 operations per cycle.

# Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation

- Mechanisms of dynamic scheduling, multiple issue, and speculation

- Extend Tomasulo's algorithm to support multiple issue superscalar pipeline with separate integer, load/store, and floating-point units (both FP multiply and FP add), each of which can initiate an operation on every clock.

# Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation



# Example

- Consider the execution of the following loop, which increments each element of an integer array, on a two-issue processor, once without speculation and once with speculation:

```
Loop: LD R2,0(R1)      ;R2=array element
      DADDIU R2,R2,#1  ;increment R2
      SD R2,0(R1)      ;store result
      DADDIU R1,R1,#8  ;increment pointer
      BNE R2,R3,LOOP   ;branch if not last element
```

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table for the first three iterations of this loop for both processors. Assume that up to two instructions of any type can commit per clock.

# Example

| Iteration number | Instructions | Issues at clock cycle number | Executes at clock cycle number | Memory access at clock cycle number | Write CDB at clock cycle number | Comment |
|---|---|---|---|---|---|---|
| 1 | LD R2,0(R1) | 1 | 2 | 3 | 4 | First issue |
| 1 | DADDIU R2,R2,#1 | 1 | 5 | | 6 | Wait for LW |
| 1 | SD R2,0(R1) | 2 | 3 | 7 | | Wait for DADDIU |
| 1 | DADDIU R1,R1,#8 | 2 | 3 | | 4 | Execute directly |
| 1 | BNE R2,R3,LOOP | 3 | 7 | | | Wait for DADDIU |
| 2 | LD R2,0(R1) | 4 | 8 | 9 | 10 | Wait for BNE |
| 2 | DADDIU R2,R2,#1 | 4 | 11 | | 12 | Wait for LW |
| 2 | SD R2,0(R1) | 5 | 9 | 13 | | Wait for DADDIU |
| 2 | DADDIU R1,R1,#8 | 5 | 8 | | 9 | Wait for BNE |
| 2 | BNE R2,R3,LOOP | 6 | 13 | | | Wait for DADDIU |
| 3 | LD R2,0(R1) | 7 | 14 | 15 | 16 | Wait for BNE |
| 3 | DADDIU R2,R2,#1 | 7 | 17 | | 18 | Wait for LW |
| 3 | SD R2,0(R1) | 8 | 15 | 19 | | Wait for DADDIU |
| 3 | DADDIU R1,R1,#8 | 8 | 14 | | 15 | Wait for BNE |
| 3 | BNE R2,R3,LOOP | 9 | 19 | | | Wait for DADDIU |

**The time of issue, execution, and writing result for a dual-issue version of pipeline *without* speculation**

# Example

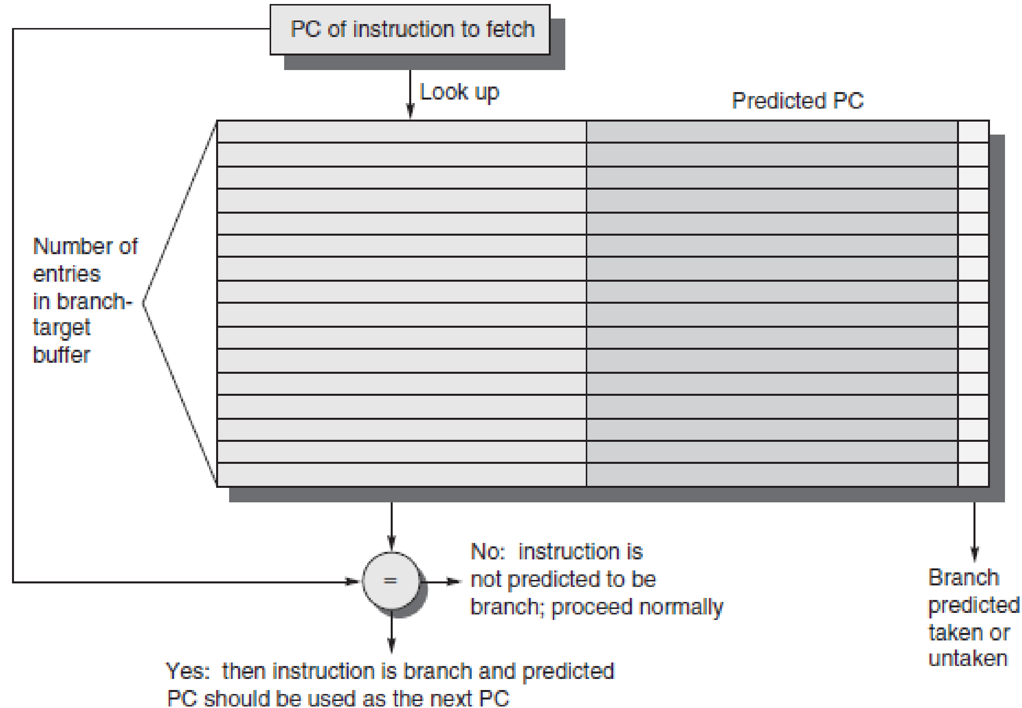
| Iteration number | Instructions | Issues at clock number | Executes at clock number | Read access at clock number | Write CDB at clock number | Commits at clock number | Comment |
|---|---|---|---|---|---|---|---|
| 1 | LD R2,0(R1) | 1 | 2 | 3 | 4 | 5 | First issue |
| 1 | DADDIU R2,R2,#1 | 1 | 5 | | 6 | 7 | Wait for LW |
| 1 | SD R2,0(R1) | 2 | 3 | | | 7 | Wait for DADDIU |
| 1 | DADDIU R1,R1,#8 | 2 | 3 | | 4 | 8 | Commit in order |
| 1 | BNE R2,R3,LOOP | 3 | 7 | | | 8 | Wait for DADDIU |
| 2 | LD R2,0(R1) | 4 | 5 | 6 | 7 | 9 | No execute delay |
| 2 | DADDIU R2,R2,#1 | 4 | 8 | | 9 | 10 | Wait for LW |
| 2 | SD R2,0(R1) | 5 | 6 | | | 10 | Wait for DADDIU |
| 2 | DADDIU R1,R1,#8 | 5 | 6 | | 7 | 11 | Commit in order |
| 2 | BNE R2,R3,LOOP | 6 | 10 | | | 11 | Wait for DADDIU |
| 3 | LD R2,0(R1) | 7 | 8 | 9 | 10 | 12 | Earliest possible |
| 3 | DADDIU R2,R2,#1 | 7 | 11 | | 12 | 13 | Wait for LW |
| 3 | SD R2,0(R1) | 8 | 9 | | | 13 | Wait for DADDIU |
| 3 | DADDIU R1,R1,#8 | 8 | 9 | | 10 | 14 | Executes earlier |
| 3 | BNE R2,R3,LOOP | 9 | 13 | | | 14 | Wait for DADDIU |

**The time of issue, execution, and writing result for a dual-issue version of pipeline *with* speculation**

# Branch Target Buffers

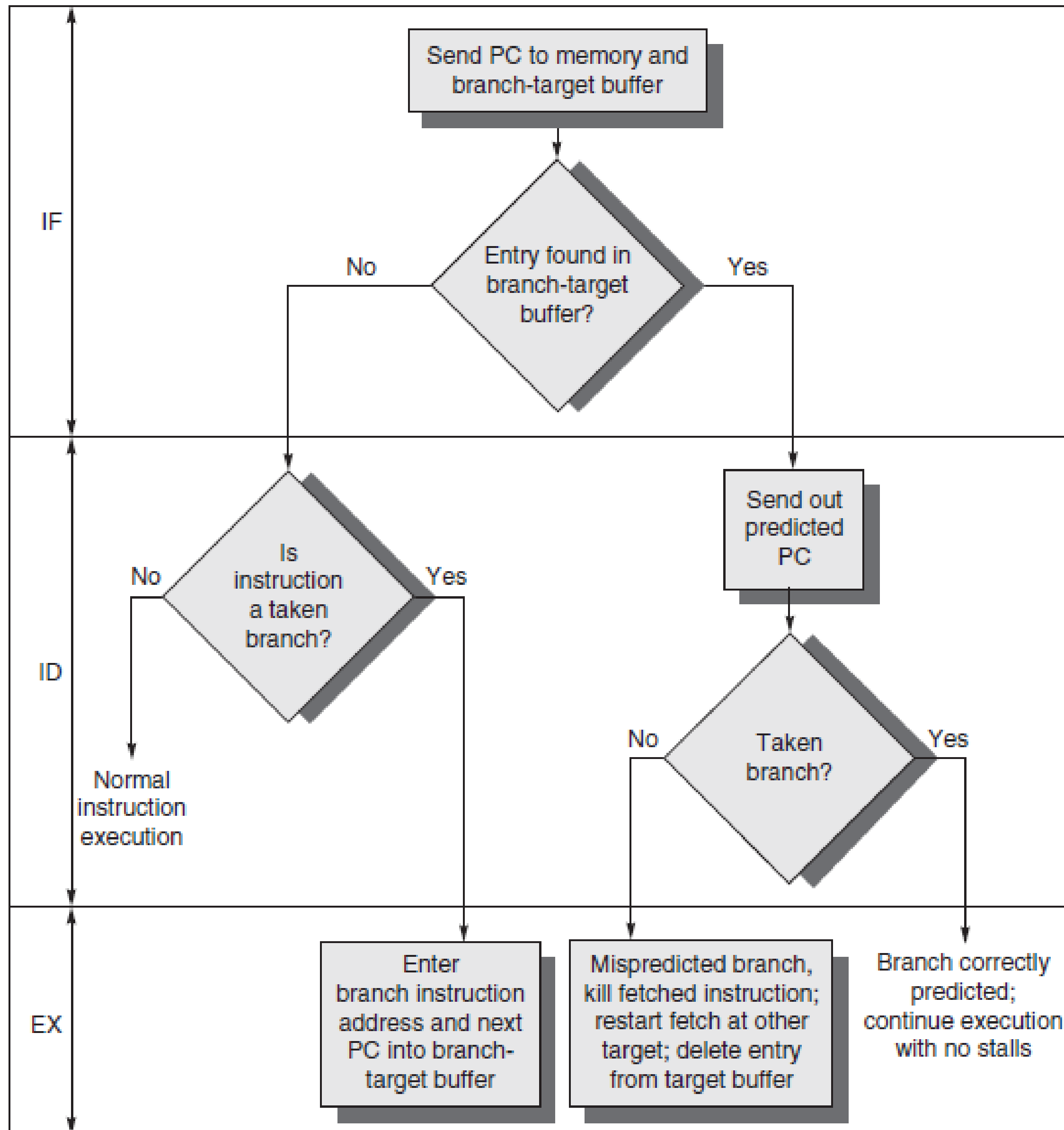
- To reduce the branch penalty for simple five-stage pipeline, as well as for deeper pipelines, it must know whether the as-yet-undecoded instruction is a branch and, if so, what the next program counter (PC) should be.
- If the instruction is a branch and next PC is known, branch penalty is zero.
- A branch-prediction cache stores the predicted address for the next instruction after a branch is called a branch-target buffer or branch-target cache.
- Branch-target buffer predicts the next instruction address and will send it out before decoding the instruction.
- If the PC of the fetched instruction matches an address in the prediction buffer, then the corresponding predicted PC is used as the next PC

# Branch Target Buffers




- The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches.
- If the PC matches one of these entries, then the instruction being fetched is a taken branch, and the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address.
- The third field, which is optional, may be used for extra prediction state bits.

# Branch Target Buffers



# Limits of ILP

# Limits of ILP

Ideal Processor - Assumptions:

- Register Renaming
  - Presence of infinite number of virtual registers.
  - Unbounded number of instructions can begin execution simultaneously.
- Branch Prediction
  - Branch prediction is perfect.
  - All conditional branches are predicted correctly.
- Jump Prediction
  - All jumps are perfectly predicted.
  - Equivalent to a processor with perfect speculation.
  - Unbounded buffer available.
- Memory Address alias Analysis
  - All memory addresses are known exactly.
  - Perfect address alias analysis (load can be moved before a store provided the addresses are not identical).
- Perfect Caches
  - All memory accesses take 1 clock cycle.

# Limits of ILP

- On Window Size and Maximum Issue Count
- Effects of Realistic Branch and Jump Prediction
- Effects of Finite Registers
- Effects of Imperfect Alias Analysis

# Window Size & Maximum Issue Count

- Perfect processor
  - Determine a set of instructions to issue, predict the branches
  - Rename all the registers
  - Check for data dependences
  - Memory dependences
  - Provide enough replicated functional units
- Set of instructions examined for simultaneous execution is called a window.
- Number of comparisons every clock = max completion rate * Window Size * No. of operands per instruction.
- Total window is limited by Required storage, comparisons, limited issue rate
- Window size directly limits the number of instructions for execution.
- Possible implementation constraints in a multiple-issue processor is large – limits the ILP.

# Effects of Branch and Jump Prediction

- Ideal Processor – jumps and branches are predicted correctly.
- Realistic prediction is not always correct !!!!

# Effect of Finite Registers

- To exploit ILP, large number of registers are required to avoid the hazards.
- Register renaming is critical – infinite number of registers is not practical.

# Effects of Imperfect Alias Analysis