1 – Amdahl’s law Revisited

Recall that earlier in the course we discussed Amdahl’s law and stated that it was originally intended for computing the speedup of a parallel computation as follows:

\[
\text{Speedup} = \frac{1}{(1 - \text{Fraction}_{\text{Parallel}}) + \frac{\text{Fraction}_{\text{Parallel}}}{\text{Speedup}_{\text{Parallel}}}}
\]

2 – Intro to ILP

Instruction Level Parallelism (ILP) improves program throughput by allowing instructions within a sequence execute concurrently.

ILP includes pipelining, but is a broader topic. Exploiting ILP uses a combined compiler design and architectural support.

3 – In Class Exercise

From our prior analysis we saw that branch instructions tended to comprise about 15% of the instructions in most programs. Suppose that the average basic block is therefore 6 instructions long, and that about 40% of the basic blocks have can only have 3 instructions parallelized (due to dependencies and hazards), while the rest can have the full 6 instructions parallelized. You can assume that other hazards have a negligible impact.

What is the maximum speedup of this system from parallel execution of basic blocks assuming that the parallel execution units run at the same speed as the sequential unit to be replaced?

4 – Solution

Recall Amdahl’s law states:

\[
\text{Speedup} = \frac{1}{(1 - \text{Fraction}_{\text{Parallel}}) + \frac{\text{Fraction}_{\text{Parallel}}}{\text{Speedup}_{\text{Parallel}}}}
\]

We know that \(\text{Fraction}_{\text{Parallel}} = 0.85\) (the number of non-branch instructions) from our given, so:

\[
\text{Speedup} = \frac{1}{(1 - 0.85) + \frac{0.85}{\text{Speedup}_{\text{Parallel}}}}
\]

Now we need to estimate \(\text{Speedup}_{\text{Parallel}}\), which we can get by using a weighted average of the number of independent instructions in the basic block and then substitute for a final solution:

\[
\text{Speedup}_{\text{Parallel}} = 0.4 \times 3 + (1 - 0.4) \times 6 = 4.8
\]

\[
\text{Speedup} = \frac{1}{(1 - 0.85) + \frac{0.85}{4.8}} \approx 3.06
\]

It is possible to get much greater speedups than this.
5 - Loop Unrolling

Loop unrolling consists of constructing a basic block that contains several iterations in line of the original loop (and iterating over the inlined iterations).

Loop unrolling reduces the number of flow of control operators (branching) in the instruction stream. This both reduces the number of instructions evaluated, and lengthens the basic block size. Overly aggressive unrolling of loops increases code size, and can contain cache misses which may degrade performance.

Loop unrolling works best for loops that are executed large numbers of times and have few dependencies between iterations.

7 - An Example Loop

Consider the following loop (in C), where x is an array of double:

```
for (i = 1; i <= 1000; ++i)
    x[i] = x[i] + s;
```

To reduce the number of comparisons to non-zero values, the compiler could reorder the array traversal to be:

```
for (i = 1000; i > 0; --i)
    x[i] = x[i] + s;
```

Eliminating the array access and using pointer arithmetic on a variable pointer:

```
for (double *ptr = x[1000]; ptr > x; --ptr)
    *ptr = *ptr + s;
```

6 - Latencies of FP operations in DLX

Assume that the following FP latencies are in place:

<table>
<thead>
<tr>
<th>Producer of Result</th>
<th>User of Result</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU Op</td>
<td>FP ALU Op</td>
<td>3</td>
</tr>
<tr>
<td>Load Double</td>
<td>Store Double</td>
<td>2</td>
</tr>
<tr>
<td>Load Double</td>
<td>FP ALU Op</td>
<td>1</td>
</tr>
<tr>
<td>Load Double</td>
<td>Store Double</td>
<td>0</td>
</tr>
</tbody>
</table>

A direct translation of this loop body without scheduling would be:

- `R0 = s`
- `F2 = 0`
- `ADD R4, F0, F2` ; `F4 = ptr + s` ; 2 stalls
- `BRZ R1, #3`
- `SUB R1, F4`
- `BLNZ R1, Loop`
- `*ptr = *ptr + s;`

The stall induced by the BRNZ is the time to compute the destination of the branch.
9 – Scheduling the Loop in DLX

Using the pipeline to schedule our loop can eliminate some stalls:

Using the pipeline to schedule our loop can eliminate some stalls:

Loop: LD F0, 0(R1) ; 1 stall
      ADDD F4, F0, F2 ; 2 stalls
      SD 0(R1), F4
      LD F6, -8(R1) ; 1 stall
      ADDD F8, F6, F2 ; 2 stalls
      SD -8(R1), F8
      LD F10, -16(R1) ; 1 stall
      ADDD F12, F10, F2 ; 2 stalls
      SD -16(R1), F12
      LD F14, -24(R1) ; 1 stall
      ADDD F16, F14, F2 ; 2 stalls
      SD -24(R1), F16
      SUBI R1, R1, #32
      BNEZ R1, Loop ; 1 stall

Note that the scheduled version takes 6 cycles while the unscheduled version takes 10 cycles.

10 – Unrolling The Loop in DLX

Unrolling a loop lengthens the basic block, which can improve scheduling. Unrolling the loop 4 times yields:

Loop: LD F0, 0(R1)
      ADDD F4, F0, F2
      SD 0(R1), F4
      LD F6, -8(R1)
      ADDD F8, F6, F2
      SD -8(R1), F8
      LD F10, -16(R1)
      ADDD F12, F10, F2
      SD -16(R1), F12
      LD F14, -24(R1)
      ADDD F16, F14, F2
      SD -24(R1), F16
      SUBI R1, R1, #32
      BNEZ R1, Loop

In Class Exercise: Show where the stalls are induced.

11 – Loop Stalls

The stalls are indicated at the following locations due to instruction order.

Loop: LD F0, 0(R1) ; 1 stall
      ADDD F4, F0, F2 ; 2 stalls
      SD 0(R1), F4
      LD F6, -8(R1) ; 1 stall
      ADDD F8, F6, F2 ; 2 stalls
      SD -8(R1), F8
      LD F10, -16(R1) ; 1 stall
      ADDD F12, F10, F2 ; 2 stalls
      SD -16(R1), F12
      LD F14, -24(R1) ; 1 stall
      ADDD F16, F14, F2 ; 2 stalls
      SD -24(R1), F16
      SUBI R1, R1, #32
      BNEZ R1, Loop ; 1 stall

Can we improve the scheduling of these instructions?

12 – Scheduling the Unrolled Loop

Rescheduling the unrolled basic block allows us to eliminate much of the stalling.

Loop: LD F0, 0(R1)
      LD F6, -8(R1)
      LD F10, -16(R1)
      LD F14, -24(R1)
      ADDD F4, F0, F2
      ADDD F8, F6, F2
      ADDD F12, F10, F2
      ADDD F16, F14, F2
      SD 0(R1), F4
      SD -8(R1), F8
      SUBI R1, R1, #32
      SD -16(R1), F12
      BNEZ R1, Loop
      SD 8(R1), F16 ; 8-32=-24

In Class Exercise: If there are any stalls in this loop, show them.
13 – Solution and Summary

There are no stalls.

Lessons Learned include:

1. We had to know to move the SD after the SUBI and BNEZ, and adjust the SD offset.
2. Determine that unrolling the loop is feasible and useful (by dependence analysis).
3. Use different registers to accommodate the unrolling to make rescheduling easier.
4. Eliminate extra tests and branches, and adjust loop maintenance code.
5. Determine the loads and stores in the unrolled loop can be interchanged.
6. Schedule the code while preserving semantics (i.e. keeping the original dependences).
7. Since the BNEZ branch test has moved to the ID stage from the EX stage in DLX, a stall has occurred. Had the branch test stayed in the EX there would not have been a stall (but the branch delay would have been 2 cycles rather than 1).

14 – Dependences

Dependence is similar to mutual interference at the pipeline level. For the purposes of definition we assume a pipeline with adequate hardware support to run at its ideal speed (i.e. free from structural hazards).

Two instructions are independent if they can execute in parallel in a pipeline without causing stalls. Dependent instructions cannot be reordered and are not parallel.

Later, we will discuss each of three kinds of dependences:

1. Data Dependence
2. Name Dependence
3. Control Dependence

15 – Data Dependence

Instruction $j$ is data dependent on a prior instruction $i$ if either:

1. $i$ produces a result used by $j$, or
2. $j$ is data dependent on $k$ and $k$ is data dependent on $i$

This implies:

1. Data dependence is transitive (i.e. you can chain them together).
2. Parallel execution would expose a chain of one or more RAW hazards.

The book uses dark diagonal downward arrows to indicate data dependencies.
16 – Removing Data Dependence

See Page 231 in the book.
The SUBI instructions in the unrolled (but not rescheduled) loop are eliminated by the compiler's recognizing the dependence.

17 – Name Dependence

A name in this case refers to a particular memory location or register used by two instructions.

Name dependence refers to two instructions using the same name without a flow of information between them through the name.

More formally, let \( i \) be an instruction prior to \( j \), then \( j \) is name dependent on \( i \) if either:

1. \( j \) writes a name that \( i \) reads, creating an antidependence which corresponds to a WAR hazard.
2. \( j \) writes to a name that \( i \) writes to, creating an output dependence which corresponds to a WAW hazard.

18 – Avoiding Name Dependence

The book shows these using light gray downward diagonal arrows. (See Page 232).

Register renaming avoids name dependence of registers, in unrolled loops by allocating a new set of registers for each iteration of the unrolled loop.

Either the compiler can do it statically or the hardware can do it (by bank switching between register sets).

19 – Control Dependence

Control dependence is used to ensure that statements following a branch are evaluated when the condition controlling the branch has an appropriate value.

Consider the example:

```c
if (p1) {
    S1;
}
if (p2) {
    S2;
}
```

S1 is control dependent on p1 and S2 is control dependent on p2 but not on p1.
20 – Control Dependence

Control dependence restricts reordering as follows:

1. An instruction that is control dependent on a branch cannot be moved before the branch.
2. An instruction that is not control dependent cannot be moved after the branch (so that the branch controls its execution).

21 – Pipelining and Control Dependence

Simple pipelines (like in Chapter 3) preserve control dependence by enforcing two important properties:

1. Instructions execute in order, so statements prior to a branch execute before the branch.
2. Control hazards detection ensures that statements after a branch execute only when it is known if the branch is taken or not.

The control dependence requirements can be relaxed if the following semantic requirements can still be satisfied:

1. Exception Behavior — The instruction order should not impact the raising of exceptions by the program.
2. Data Flow — The data invariants of the program must be maintained (i.e. preconditions and postconditions).

22 – Control Dependence - Example Code

Exception behavior can prevent reordering:

BEQZ R2, L1; Must come before the LW
LW R1, (R2); Could access an invalid address

Data flow requirements may prevent reordering too:

ADD R1, R2, R3; Sets R1 prior to branch
BEQZ R4, L, R5; Modifies R1 in branch
SUB R1, R7, R8; R1 affected by whether L:
GR R1, R7, R8; R1 take or not

The branch is take or not
24 - Loop Level Parallelism

Recall our example:

```c
for (i = 1; i <= 1000; ++i)
    x[i] = x[i] + s;
```

Here there is only dependence on `x[i]` within the iteration, not between iterations. We can execute the iterations in parallel (due to their mutual non-interference).

25 - Loop Carried Dependence

Suppose instead we have something like:

```c
for (i = 1; i <= 100; ++i){
    A[i+1] = A[i] + C[i];    // S1
    B[i+1] = B[i] + A[i + 1]; // S2
}
```

We can see that:

1. S1 uses `A[i]` which is computed by S1 in an earlier iteration and S2 is similar with respect to `B[i]`. This is a loop carried dependence.
2. S2 uses `A[i+1]` computed by S1 in the same iteration.

Unfortunately the dependence of S1 on itself forces successive evaluations of S1 to be executed sequentially.

26 - Dynamic Scheduling

We discussed static instruction scheduling, which was done by compiler support. Basically the compiler detected hazard conditions and reordered the instructions using information about the program semantics.

Dynamic Scheduling refers to the hardware detecting hazards at run time and reordering the (partial) evaluations to avoid stalling. This is especially helpful since it is impossible for the compiler to detect some hazards (e.g. structural hazards).

To support this the ID pipe stage will need to split into:

1. Issue — Decode instruction and check for structural hazards.
2. Read Operands — Wait until no hazard and then read operands.

In general instructions will be issued in the order in which they appear in the instruction stream, but the partial evaluation will be computed out of order.
27 – Dynamic Scheduling Challenges

Consider the example:
DIVD F0, F2, F4 ; takes 25 cycles
ADDD F10, F0, F8 ; F0 depends on DIVD
SUBD F8, F8, F14 ; F8 antidependent on ADDD

So reordering this sequence of instructions is difficult, however there is considerable stall cycles induced. In fact the DIVD would induce 24 stall cycles and the ADDD would induce 3 more stall cycles in the worst case.

28 – Score Boarding

The CDC 6600 was one of the first super computers, (designed by Seymour Cray) to support dynamic scheduling using a scoreboard.

Informally the score board is designed to keep track of:
1. Instructions in various stages of execution and

This permits the hardware to detect hazards and to complete them.

29 – Stages of Execution in Score Boarding

Instead of the traditional pipeline stages in DLX, we substitute:
1. Issue
2. Read Operands
3. Execution
4. Write Result

30 – Instruction Issue in Score Boarding

`Issue(IS)` replaces the first part of the ID stage.

An instruction can be issued if:
1. A suitable functional unit is free (avoiding structural hazards) and
2. No active instruction has the same destination register (to avoid WAW hazards).

If the instruction cannot be issued it `stalls` and no further instruction issues will occur until the hazard clears.

The scoreboard issues the instruction to the functional unit and updates its state.
31 – Read Operands in Score Boarding

Read Operands (RO) replaces the second part of the ID stage.

An instruction can read its operands if:
1. No earlier issued instruction writes an operand or
2. The register containing the operand is being written to by an active FU.

This procedure avoids RAW hazards but allows out of order execution.

When the operands are available, the scoreboard allows the FU to read the operands from the registers and begin execution.

32 – Execution in Score Boarding

Execution (EX) replaces the EX stage.

An instruction can execute as soon as the FU receives its operands.

When the result is ready the scoreboard is notified of completion of execution

EX requires multiple cycles for the DLX FP pipeline.

33 – Write Results in Score Boarding

Write Results (WR) replaces the WB stage.

An instruction can write its results to the user visible register file when:
1. all prior instructions (in order of issue) have read their operands and
2. no prior instruction has an operand in the same register as the result of the completing instruction

This avoids WAR hazards.

34 – Component of the Scoreboard

The Scoreboard is composed of:

1. **Instruction Status** — Which step the instruction is in.

2. **Functional Unit Status** — Indicates the state of each FU with the fields:
   (a) Busy — Is the FU in use?
   (b) Op — Which operation to perform?
   (c) Fi — Destination Register
   (d) Fj, Fk — Source-register numbers
   (e) Qi, Qk — Functional units producing Fj and Fk.
   (f) Rj,Rk — Flags indicating when Fj, Fk are ready.

3. **Register result status** — Indicates which FU will write each register when an active instruction as the register as its destination.
35 – A Scoreboarding Example

Consider the DLX sequence:

- LD F6, 34(R2)
- LD F2, 45(R3)
- MULT F0, F2, F4
- SUBD F8, F6, F2
- DIVD F10, F0, F6
- ADDD F6, F8, F2

On a DLX machine with the following FUs (timings are not necessarily realistic):

1. An integer unit (handles the LDs) (1 Cycle)
2. Two FP multipliers, Mult1, and Mult2 (10 stage pipeline)
3. An FP Adder (2 stage pipeline)
4. An FP divider (25 Cycle unpipelined)

For convenience we will make the (somewhat unrealistic) assumption that any register read by an instruction which is not loaded first contains appropriate values and that all units are initially available.

36 – A DLX Scoreboard Architecture

With a scoreboard, the DLX architecture considered looks like:

![Diagram of DLX scoreboard architecture](image)

**FIGURE 4.3** The basic structure of a DLX processor with a scoreboard.

36.1 – Setting Up Scoreboard Notation

We make a table for each component of the scoreboard, assuming the machine is idle prior to starting. The FU table has the additional “field” \( t \) for time until an executing operation finishes. This is not really in the scoreboard:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>( j )</th>
<th>( k )</th>
<th>( t )</th>
<th>( RO )</th>
<th>( RX )</th>
<th>( WR )</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6, 34(R2)</td>
<td>34</td>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F2, 45(R3)</td>
<td>45</td>
<td>F3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULT F0, F2, F4</td>
<td>F2</td>
<td>F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBD F8, F6, F2</td>
<td>F6</td>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVD F10, F0, F6</td>
<td>F0</td>
<td>F6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F6, F8, F2</td>
<td>F6</td>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FU status**

<table>
<thead>
<tr>
<th>Name</th>
<th>( t )</th>
<th>( Busy )</th>
<th>Op</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intit</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtle</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register Read Status**

<table>
<thead>
<tr>
<th>F0</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Integer registers are omitted since they are not written to.
36.2 – The Example — Cycle 1

Issue the first LD

Instruction Status

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>RO</th>
<th>EX</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD P6, 3(R2)</td>
<td>34</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>LD P2, 4(R3)</td>
<td>45</td>
<td>R3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MULT P6, P2, P4</td>
<td>P2</td>
<td>P4</td>
<td>P2</td>
<td>P4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>SUBD P6, P6, P2</td>
<td>P6</td>
<td>P2</td>
<td>P6</td>
<td>P2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>DIVD P6, P6, P2</td>
<td>P6</td>
<td>P2</td>
<td>P6</td>
<td>P2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ADDD P6, P6, P2</td>
<td>P6</td>
<td>P2</td>
<td>P6</td>
<td>P2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

FU Status

<table>
<thead>
<tr>
<th>Name</th>
<th>t</th>
<th>Busy</th>
<th>Op</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>Q1</th>
<th>Q2</th>
<th>R1</th>
<th>Rk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integ</td>
<td>Y</td>
<td>LD</td>
<td>P6</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult1</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult2</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Status

<table>
<thead>
<tr>
<th>FU</th>
<th>integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>F2</td>
</tr>
<tr>
<td>F4</td>
<td>F6</td>
</tr>
<tr>
<td>F8</td>
<td>F10</td>
</tr>
</tbody>
</table>

36.3 – The Example — Cycle 2

The Second LD cannot be issued due to a structural hazard (the First LD is using the Integer FU).

Instruction Status

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>RO</th>
<th>EX</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD P6, 3(R2)</td>
<td>34</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>LD P2, 4(R3)</td>
<td>45</td>
<td>R3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MULT P6, P2, P4</td>
<td>P2</td>
<td>P4</td>
<td>P2</td>
<td>P4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>SUBD P6, P6, P2</td>
<td>P6</td>
<td>P2</td>
<td>P6</td>
<td>P2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>DIVD P6, P6, P2</td>
<td>P6</td>
<td>P2</td>
<td>P6</td>
<td>P2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ADDD P6, P6, P2</td>
<td>P6</td>
<td>P2</td>
<td>P6</td>
<td>P2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

FU Status

<table>
<thead>
<tr>
<th>Name</th>
<th>t</th>
<th>Busy</th>
<th>Op</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>Q1</th>
<th>Q2</th>
<th>R1</th>
<th>Rk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integ</td>
<td>Y</td>
<td>LD</td>
<td>P6</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult1</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult2</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Status

<table>
<thead>
<tr>
<th>FU</th>
<th>integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>F2</td>
</tr>
<tr>
<td>F4</td>
<td>F6</td>
</tr>
<tr>
<td>F8</td>
<td>F10</td>
</tr>
</tbody>
</table>

36.4 – The Example — Cycle 3

The Second LD cannot be issued due to a structural hazard (the First LD is using the Integer FU).

Instruction Status

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>RO</th>
<th>EX</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD P6, 3(R2)</td>
<td>34</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>LD P2, 4(R3)</td>
<td>45</td>
<td>R3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MULT P6, P2, P4</td>
<td>P2</td>
<td>P4</td>
<td>P2</td>
<td>P4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>SUBD P6, P6, P2</td>
<td>P6</td>
<td>P2</td>
<td>P6</td>
<td>P2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>DIVD P6, P6, P2</td>
<td>P6</td>
<td>P2</td>
<td>P6</td>
<td>P2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ADDD P6, P6, P2</td>
<td>P6</td>
<td>P2</td>
<td>P6</td>
<td>P2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

FU Status

<table>
<thead>
<tr>
<th>Name</th>
<th>t</th>
<th>Busy</th>
<th>Op</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>Q1</th>
<th>Q2</th>
<th>R1</th>
<th>Rk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integ</td>
<td>Y</td>
<td>LD</td>
<td>P6</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult1</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult2</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Status

<table>
<thead>
<tr>
<th>FU</th>
<th>integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>F2</td>
</tr>
<tr>
<td>F4</td>
<td>F6</td>
</tr>
<tr>
<td>F8</td>
<td>F10</td>
</tr>
</tbody>
</table>

36.5 – The Example — Cycle 4

The Second LD cannot be issued due to a structural hazard (the First LD is using the Integer FU).

Instruction Status

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>RO</th>
<th>EX</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD P6, 3(R2)</td>
<td>34</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>LD P2, 4(R3)</td>
<td>45</td>
<td>R3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MULT P6, P2, P4</td>
<td>P2</td>
<td>P4</td>
<td>P2</td>
<td>P4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>SUBD P6, P6, P2</td>
<td>P6</td>
<td>P2</td>
<td>P6</td>
<td>P2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>DIVD P6, P6, P2</td>
<td>P6</td>
<td>P2</td>
<td>P6</td>
<td>P2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ADDD P6, P6, P2</td>
<td>P6</td>
<td>P2</td>
<td>P6</td>
<td>P2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

FU Status

<table>
<thead>
<tr>
<th>Name</th>
<th>t</th>
<th>Busy</th>
<th>Op</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>Q1</th>
<th>Q2</th>
<th>R1</th>
<th>Rk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integ</td>
<td>Y</td>
<td>LD</td>
<td>P6</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult1</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult2</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 36.8 – The Example — Cycle 7

The SUBD can be issued. Instruction #12.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination</th>
<th>Source 1</th>
<th>Source 2</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBD 1000</td>
<td>A</td>
<td>B</td>
<td>12</td>
<td>SUBD</td>
</tr>
</tbody>
</table>

### 36.7 – The Example — Cycle 6

The MUTD can be issued, since its structural hazard has cleared.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination</th>
<th>Source 1</th>
<th>Source 2</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUTD 2000</td>
<td>A</td>
<td>B</td>
<td>13</td>
<td>MUTD</td>
</tr>
</tbody>
</table>

### 36.6 – The Example — Cycle 5

The second ID can now be issued since the structural hazard is cleared.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination</th>
<th>Source 1</th>
<th>Source 2</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD 3000</td>
<td>A</td>
<td>B</td>
<td>14</td>
<td>ADD</td>
</tr>
</tbody>
</table>

### 36.9 – The Example — Cycle 8

The DIVP can be issued. MUTD and SUBD hazards are cleared.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination</th>
<th>Source 1</th>
<th>Source 2</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVP 4000</td>
<td>A</td>
<td>B</td>
<td>15</td>
<td>DIVP</td>
</tr>
</tbody>
</table>

### Table 3.1

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination</th>
<th>Source 1</th>
<th>Source 2</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

36 = A blank word address. 37 = Right of computer bus. 38 = Any address.
36.12 - The Example - Cycle 12

SUBD Completes its WR stage.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>RO</th>
<th>EX</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6, 34(R2)</td>
<td>34</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>LD F2, 45(R3)</td>
<td>45</td>
<td>R3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>MULTD F6, F2, F3</td>
<td>F6</td>
<td>F2</td>
<td>F3</td>
<td>6</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>SUBD F8, F6, F2</td>
<td>F8</td>
<td>F6</td>
<td>F2</td>
<td>7</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>ADDD F6, F8, F2</td>
<td>F6</td>
<td>F8</td>
<td>F2</td>
<td>8</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

FU Status

<table>
<thead>
<tr>
<th>Name</th>
<th>t</th>
<th>Busy</th>
<th>Op</th>
<th>F1</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Y</td>
<td>MULTD</td>
<td>F6</td>
<td>F2</td>
<td>F4</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mdh1</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mdh2</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Status

- F6 F2 F4 F6 F8 F10
- PU Mdh1
- Add divide

36.10 - The Example - Cycle 9

The DIVD stalls waiting for the MULTD.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>RO</th>
<th>EX</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6, 34(R2)</td>
<td>34</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>LD F2, 45(R3)</td>
<td>45</td>
<td>R3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>MULTD F6, F2, F3</td>
<td>F6</td>
<td>F2</td>
<td>F3</td>
<td>6</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>SUBD F8, F6, F2</td>
<td>F8</td>
<td>F6</td>
<td>F2</td>
<td>7</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>ADDD F6, F8, F2</td>
<td>F6</td>
<td>F8</td>
<td>F2</td>
<td>8</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

FU Status

<table>
<thead>
<tr>
<th>Name</th>
<th>t</th>
<th>Busy</th>
<th>Op</th>
<th>F1</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mdh1</td>
<td>10</td>
<td>Y</td>
<td>MULTD</td>
<td>F6</td>
<td>F2</td>
<td>F4</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Mdh2</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>2</td>
<td>Y</td>
<td>SUBD</td>
<td>F8</td>
<td>F6</td>
<td>F2</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>Y</td>
<td>DIVD</td>
<td>F10</td>
<td>F6</td>
<td>F6</td>
<td>F6</td>
<td>Mdh1</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Register Status

- F6 F2 F4 F6 F8 F10
- PU Mdh1
- Add divide

36.13 - The Example - Cycle 13

The ADDD clears its structural hazard and can be issued.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>RO</th>
<th>EX</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6, 34(R2)</td>
<td>34</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>LD F2, 45(R3)</td>
<td>45</td>
<td>R3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>MULTD F6, F2, F3</td>
<td>F6</td>
<td>F2</td>
<td>F3</td>
<td>6</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>SUBD F8, F6, F2</td>
<td>F8</td>
<td>F6</td>
<td>F2</td>
<td>7</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>ADDD F6, F8, F2</td>
<td>F6</td>
<td>F8</td>
<td>F2</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FU Status

<table>
<thead>
<tr>
<th>Name</th>
<th>t</th>
<th>Busy</th>
<th>Op</th>
<th>F1</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mdh1</td>
<td>6</td>
<td>Y</td>
<td>MULTD</td>
<td>F6</td>
<td>F2</td>
<td>F4</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Mdh2</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Y</td>
<td>ADDD</td>
<td>F6</td>
<td>F8</td>
<td>F2</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>Y</td>
<td>DIVD</td>
<td>F10</td>
<td>F6</td>
<td>F6</td>
<td>Mdh1</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

Register Status

- F6 F2 F4 F6 F8 F10
- PU Mdh1
- Add divide

36.11 - The Example - Cycle 11

SUBD Completes its EX stage.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>RO</th>
<th>EX</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6, 34(R2)</td>
<td>34</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>LD F2, 45(R3)</td>
<td>45</td>
<td>R3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>MULTD F6, F2, F3</td>
<td>F6</td>
<td>F2</td>
<td>F3</td>
<td>6</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>SUBD F8, F6, F2</td>
<td>F8</td>
<td>F6</td>
<td>F2</td>
<td>7</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>ADDD F6, F8, F2</td>
<td>F6</td>
<td>F8</td>
<td>F2</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FU Status

<table>
<thead>
<tr>
<th>Name</th>
<th>t</th>
<th>Busy</th>
<th>Op</th>
<th>F1</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mdh1</td>
<td>8</td>
<td>Y</td>
<td>MULTD</td>
<td>F6</td>
<td>F2</td>
<td>F4</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Mdh2</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Y</td>
<td>SUBD</td>
<td>F8</td>
<td>F6</td>
<td>F2</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>Y</td>
<td>DIVD</td>
<td>F10</td>
<td>F6</td>
<td>F6</td>
<td>F6</td>
<td>Mdh1</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

Register Status

- F6 F2 F4 F6 F8 F10
- PU Mdh1
- Add divide
### 36.16 - The Example — Cycle 17

**ADDD stalls because of WAR hazard (with DIVD).**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>RO</th>
<th>EX</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0, #34</td>
<td>R3</td>
<td>34</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>LD F2, 34(R3)</td>
<td>R3</td>
<td>45</td>
<td>R3</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>MultD F0, F1, F2, F4</td>
<td>F2</td>
<td>F4</td>
<td>F2</td>
<td>6</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>SubD F6, F6, F2</td>
<td>F6</td>
<td>F2</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>DivD F10, F6, F2</td>
<td>F10</td>
<td>F6</td>
<td>F2</td>
<td>13</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

**Register Stage**

- FU Mult1
- Add
- divide

**FU Status**

- F0, F2, F4, F6, F8, F10

### 36.14 - The Example — Cycle 14

**ADDD can RO and begin execution stage.**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>RO</th>
<th>EX</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0, #34</td>
<td>R0</td>
<td>34</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>LD F2, 34(R3)</td>
<td>R3</td>
<td>45</td>
<td>R3</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>MultD F0, F2, F4</td>
<td>F2</td>
<td>F4</td>
<td>F2</td>
<td>6</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>SubD F6, F6, F2</td>
<td>F6</td>
<td>F2</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>DivD F10, F6, F2</td>
<td>F10</td>
<td>F6</td>
<td>F2</td>
<td>13</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

**Register Stage**

- FU Mult1
- Add
- divide

### 36.17 - The Example — Cycle 19

**MULTD completes its EX stage.**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>RO</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0, #34</td>
<td>R0</td>
<td>34</td>
<td>R2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>LD F2, 34(R3)</td>
<td>R3</td>
<td>45</td>
<td>R3</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>MultD F0, F2, F4</td>
<td>F2</td>
<td>F4</td>
<td>F2</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>SubD F6, F6, F2</td>
<td>F6</td>
<td>F2</td>
<td>7</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>DivD F10, F6, F2</td>
<td>F10</td>
<td>F6</td>
<td>F2</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

**Register Stage**

- FU Mult1
- Add
- divide

### 36.15 - The Example — Cycle 16

**ADDD completes its EX stage.**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>RO</th>
<th>EX</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0, #34</td>
<td>R0</td>
<td>34</td>
<td>R2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>LD F2, 34(R3)</td>
<td>R3</td>
<td>45</td>
<td>R3</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>MultD F0, F2, F4</td>
<td>F2</td>
<td>F4</td>
<td>F2</td>
<td>6</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>SubD F6, F6, F2</td>
<td>F6</td>
<td>F2</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>DivD F10, F6, F2</td>
<td>F10</td>
<td>F6</td>
<td>F2</td>
<td>13</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

**Register Stage**

- FU Mult1
- Add
- divide
36.20 – The Example — Cycle 22

The ADDD clears its WAR hazard and can do its WR stage.

![Table showing Instruction and FU Status]

36.21 – The Example — Cycle 61

The DIVD completes its EX stage.

![Table showing Instruction and FU Status]

36.18 – The Example — Cycle 20

MULTD does its WR stage, making F2 available for DIVD.

![Table showing Instruction and FU Status]

36.19 – The Example — Cycle 21

Finally the DIVD clears its hazard and can RO.

![Table showing Instruction and FU Status]
37 – Scoreboarding Performance

Scoreboarding has moved from supercomputers into microprocessors (e.g. PowerPC 620).
The CDC 6600 had (in 1963 mind you):
1. 16 FUs (4 FP units, 5 Memory Access Units, and 7 integer Units).
2. A data trunk was allocated to each group of 4 FUs so redundant buses to the data file were available (avoiding a bottleneck).
3. No compiler support for pipeline scheduling.

The designers measured a speedup of 1.7 for FORTRAN programs and 2.5 for hand coded assembly.

Ungraded homework: how long would it take to run the example through a DLX pipeline in the absence of scoreboards (you can assume no other hazards, and forwarding is available).

38 – Summary of Scoreboarding

A scoreboard:
1. Centralizes the scheduling of dynamic instructions via the scoreboard.
2. Has limited ability to eliminate stalls due to:
   (a) Limits in the parallelism in the instructions.
   (b) The number of scoreboard entries (which determines the number of candidate instructions to run, called the window, for the pipeline).
   (c) The number of and type of functional units (structural hazards).
   (d) The presence of antidependences and output dependences (WAR and WAW stalls).
40 – Architecture for Tomasulo’s Algorithm

The book applies Tomasulo’s algorithm to DLX FP by assuming that there are is one FU per reservation station (2 multipliers/dividers and 4 adders). Execution control tables are omitted.

![Diagram of DLX FP using Tomasulo's algorithm](image)

**FIGURE 4.8 The basic structure of a DLX FP unit using Tomasulo’s algorithm.**

41 – Pipeline Stages of Tomasulo’s Algorithm

In DLX we make the following replacements to make the FP pipeline work:

1. Issue — Get an instruction from the FP operation queue.
   - (a) If the instruction is an issue
   - (b) If the instruction is a load or store and there is an available buffer, issue it. Otherwise stall on the structural hazard

2. Execute — Executes the operation and detects RAW hazards:
   - (a) If an operand is unavailable, monitor the CDB pending the computation of the register’s contents.
   - (b) When the operand becomes available, place it in the corresponding reservation station.
   - (c) When both operands are available, execute the operation.

42 – Tomasulo’s Algorithm Vs. Scoreboarding

In Tomasulo’s algorithm differs from scoreboard because:

1. WAW and WAR hazards are eliminated by register renaming during instruction issue.
2. The CDB broadcasts results rather than having a centralized scoreboard wait for registers.
3. The loads and stores are treated as functional units.
43 – Reservation Table Structure

Each reservation station has the following fields:

- **OP** — The operation to perform on source operands S1 and S2.
- **Qj,Qk** — The reservations producing the source operand, a value of 0 means the source operand is either already available in Vj,Vk or is unnecessary.
- **Vj,Vk** — The value of the source operands. Note either the Q or V field is valid for an operand but not both.
- **Busy** — the reservation station and its FU are occupied.

44 – Registers and Load/Store Buffers

Each Register File and store buffer has a field:

- **Qi** — The number of the reservation station that will receive the result of the operation. If Qi is blank, there is no computation pending which will store its result in this register or buffer. (For a register, that means the value is already stored in the register).

Load and store buffers have a Busy field, to signal buffer availability.

The store buffers have a V field containing the value to store.

45 – An Example

Consider the instruction sequence:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source 1</th>
<th>Source 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6, 34(R2)</td>
<td>F6</td>
<td>34</td>
</tr>
<tr>
<td>LD F2, 45(R3)</td>
<td>F2</td>
<td>45</td>
</tr>
<tr>
<td>MULTD F0, F2, F4</td>
<td>F0</td>
<td>F2</td>
</tr>
<tr>
<td>SUBD F8, F6, F2</td>
<td>F8</td>
<td>F6</td>
</tr>
<tr>
<td>DIVD F10, F0, F6</td>
<td>F10</td>
<td>F0</td>
</tr>
<tr>
<td>ADDD F6, F8, F2</td>
<td>F6</td>
<td>F8</td>
</tr>
</tbody>
</table>

(which we also used for our scoreboard example).
45.2 – Tomasulo’s Algorithm, Cycle 1

45.3 – Tomasulo’s Algorithm, Cycle 2

45.4 – Tomasulo’s Algorithm, Cycle 3

LD1 is done, anything waiting for it?

45.5 – Tomasulo’s Algorithm, Cycle 4

LD2 is done, anything waiting for it?
45.8 - Tomasulo's Algorithm, Cycle 7

SUBD Completes its EX, anything waiting on ADD1 Results?

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Result</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>EX</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>P0</td>
<td>3</td>
<td>4</td>
<td>R2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>LD</td>
<td>P2</td>
<td>4</td>
<td>5</td>
<td>R3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>MULTD</td>
<td>P0</td>
<td>P2</td>
<td>P3</td>
<td>F4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SUBD</td>
<td>P0</td>
<td>P2</td>
<td>P2</td>
<td>F2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>DIVD</td>
<td>F10</td>
<td>F0</td>
<td>F0</td>
<td>F6</td>
<td>F6</td>
<td>5</td>
</tr>
<tr>
<td>ADDD</td>
<td>P0</td>
<td>P2</td>
<td>P2</td>
<td>F2</td>
<td>F2</td>
<td>6</td>
</tr>
</tbody>
</table>

Busy Addresses

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Vk</th>
<th>Qj</th>
<th>Qk</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Add1</td>
<td>Y</td>
<td>SUBD</td>
<td>M(34+R4)</td>
<td>M(45 + R3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Add2</td>
<td>Y</td>
<td>ADDD</td>
<td>M(45 + R3)</td>
<td>ADD1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Add3</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Mul1</td>
<td>Y</td>
<td>MULTD</td>
<td>M(45 + R3)</td>
<td>R(F4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Mul2</td>
<td>Y</td>
<td>DIVD</td>
<td>M(34 + R2)</td>
<td>MULT1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Status

FU | MULT1 | ADD2 | ADDI | MULT2

45.9 - Tomasulo's Algorithm, Cycle 8

SUBD Completes its WR

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Result</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>EX</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>P0</td>
<td>3</td>
<td>4</td>
<td>R2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>LD</td>
<td>P2</td>
<td>4</td>
<td>5</td>
<td>R3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>MULTD</td>
<td>P0</td>
<td>P2</td>
<td>P3</td>
<td>F4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SUBD</td>
<td>P0</td>
<td>P2</td>
<td>P2</td>
<td>F2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>DIVD</td>
<td>F10</td>
<td>F0</td>
<td>F0</td>
<td>F6</td>
<td>F6</td>
<td>5</td>
</tr>
<tr>
<td>ADDD</td>
<td>P0</td>
<td>P2</td>
<td>P2</td>
<td>F2</td>
<td>F2</td>
<td>6</td>
</tr>
</tbody>
</table>

Busy Addresses

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Vk</th>
<th>Qj</th>
<th>Qk</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Add1</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Add2</td>
<td>Y</td>
<td>ADDD</td>
<td>R(F2)</td>
<td>M(45 + R3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Add3</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Mul1</td>
<td>Y</td>
<td>MULTD</td>
<td>M(45 + R3)</td>
<td>R(F4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Mul2</td>
<td>Y</td>
<td>DIVD</td>
<td>M(34 + R2)</td>
<td>MULT1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Status

FU | MULT1 | ADD2 | ADDI | MULT2

45.7 - Tomasulo's Algorithm, Cycle 6

ADDD can issue (ALL instructions are ISSUED!)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Result</th>
<th>j</th>
<th>k</th>
<th>IS</th>
<th>EX</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>P0</td>
<td>3</td>
<td>4</td>
<td>R2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>LD</td>
<td>P2</td>
<td>4</td>
<td>5</td>
<td>R3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>MULTD</td>
<td>P0</td>
<td>P2</td>
<td>P3</td>
<td>F4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SUBD</td>
<td>P0</td>
<td>P2</td>
<td>P2</td>
<td>F2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>DIVD</td>
<td>F10</td>
<td>F0</td>
<td>F0</td>
<td>F6</td>
<td>F6</td>
<td>5</td>
</tr>
<tr>
<td>ADDD</td>
<td>P0</td>
<td>P2</td>
<td>P2</td>
<td>F2</td>
<td>F2</td>
<td>6</td>
</tr>
</tbody>
</table>

Busy Addresses

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Vk</th>
<th>Qj</th>
<th>Qk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Add1</td>
<td>Y</td>
<td>SUBD</td>
<td>M(34+R4)</td>
<td>M(45 + R3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Add2</td>
<td>Y</td>
<td>ADDD</td>
<td>M(45 + R3)</td>
<td>ADD1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Add3</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Mul1</td>
<td>Y</td>
<td>MULTD</td>
<td>M(45 + R3)</td>
<td>R(F4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Mul2</td>
<td>Y</td>
<td>DIVD</td>
<td>M(34 + R2)</td>
<td>MULT1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register Status

FU | MULT1 | ADD2 | ADDI | MULT2
46 - Conclusions

The Tomasulo example completed in 57 cycles vs. 62 for the scoreboard.

- Tomasulo’s algorithm had pipelined functional units and more of them (3 adders and 2 multiplier/dividers vs. 2 adders, 1 multiplier and 1 divider).
- Tomasulo’s algorithm avoids WAR and WAW hazards, while scoreboard has to stall.
- Neither scoreboard nor Tomasulo’s algorithm can issue on a structural hazard.
- Tomasulo uses broadcast of results, while scoreboard reads/writes registers.
- Tomasulo has distributed reservation stations for control vs. a centralized scoreboard.
- Tomasulo has a window size of up to 14 instructions, while scoreboard has up to 5 instructions.

Tomasulo’s algorithm has more complexity built in, and the CDB can become a bottleneck.
47 – Dynamic Hardware Prediction

Earlier we discussed using a profiler to gather statistics from previous runs of the software to do static branch prediction.

Now we are going to consider the use of hardware to predict the destination of a branch (taken or not taken) based on previous behavior of the current execution.

48 – Branch Prediction Buffers

A Branch prediction buffer:

1. Stores some (small) number of bits to encode the prediction in a hash table for each branch instruction.
2. The hash index is the low order address bits of the branch instruction.
3. The bits are aged to discard old predictions.
4. 2 Mispredictions required before prediction changes.

![Diagram showing the states in a two-bit prediction scheme.](image)

49 – Branch Prediction Buffer Performance

The effectiveness on dynamic branch prediction reflects the variance in the probability of taking the branch. Recall that for a random variable \( X \) variance is:

\[
\text{VAR} = E [(X - E[X])^2] = E [X^2 - (E[X])^2]
\]

We cannot estimate this easily, since it is a function of the program’s structure and its inputs, so we need to check experimental results on this.

50 – Branch Prediction Buffer Performance

Experimental measurements show that dynamic prediction is highly effective (with many programs getting over 90% correct).

![Bar chart showing prediction accuracy for SPEC95 benchmarks.](image)
51 - Branch Prediction Buffer Performance

Given a fixed branch prediction buffer size of 2 bits, we see that we do not get a significant win using an infinite buffer as opposed to using a 4096 entry buffer.

![Graph showing prediction accuracy of a 4096-entry two-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks.]

52 - Correlating Branch Predictors

Sometimes, the behavior of two branches interact, as seen below:

\[
\begin{align*}
\text{if} \ (d \equiv 0) \\
\quad d = 1; \\
\text{if} \ (d \equiv 1)
\end{align*}
\]

which would generate code looking like:

- BNEZ \ R1, L1 \ ; 1 Branch b1
- ADDI \ R1, R0, #1 \ ; 2 \ R1 = d = 1;
- L1: SUBI \ R3, R1, #1 \ ; 3 Test for branch 2
- BNEZ \ R3, L2 \ ; 4 Branch b2

... L2:

53 - Uncorrelated Prediction Failure

Suppose \( d \in \{0, 1, 2\} \), and \( d_i \) represents the value of \( d \) at line \( i \):

\[
\begin{array}{cccccc}
d_0 & d_1 \equiv 0 & b1 & d_3 & d_3 \equiv 1 & b2 \\
0 & Y & NT & 1 & Y & NT \\
1 & N & T & 1 & N & NT \\
2 & N & T & 2 & N & T \\
\end{array}
\]

Suppose \( d \) alternates between 2 and 0 and a one bit branch predictor is initialized to NT and \( P_1 \) is the predictor of branch \( b1 \):

\[
\begin{array}{cccccc}
d_0 & P_1 & b1 & \text{New} \ P_1 & P_2 & b2 & \text{new} \ P_2 \\
2 & NT & T & T & NT & T & T \\
0 & T & NT & NT & T & NT & NT \\
2 & NT & T & T & NT & T & T \\
0 & T & NT & NT & T & NT & NT \\
\end{array}
\]

Every Branch is mispredicted!

54 - Using Correlation in Branch Prediction

1. Behavior of nearby branches can impact behavior of the current branch, i.e. their behaviors are correlated.

2. Select the taken/not taken behavior of \( m \) most recent branches as inputs to our predictor.

3. Maintain \( n \) bits of history information per branch.

4. We use \((m, n)\) to denote selecting between \( 2^m \) previous branch outcomes, each with \( n \) bits of counter information.

So our 2 bit predictor was \((0, 2)\) and the failure example was \((0, 1)\).
55 – Using Correlation in Branch Prediction

1. Behavior of nearby branches can impact behavior of the current branch, i.e. their behaviors are correlated.

2. Select the taken/not taken behavior of \( m \) most recent branches as inputs to our predictor.

3. Maintain \( n \) bits of history information per branch.

4. We use \( (m, n) \) to denote selecting between \( 2^m \) previous branch outcomes, each with \( n \) bits of counter information.

So our 2 bit predictor was \((0,2)\) and the failure example was \((0,1)\).

56 – Correlated Branch Prediction

So the correlated branch predictor is a function \( P(A, H) \), implemented using a table lookup:

1. \( A \) — the Address of the branch in question
2. \( H \) — the History of the last \( m \) branches

![Figure 4.20](image-url)

**57 – In Class Exercise**

Suppose a branch prediction table has 4096 branch address slots. How many bits of memory are needed to store the corresponding \((4,2)\) table?

**58 – Solution**

We are given \( m = 4 \) branches of history being correlated, \( n = 2 \) bits per entry of data in the table and 4096 branch address slots. We can denote the number of bits of state information as:

\[
\text{Number of Bits} = \text{Number of Entries} \times \text{Number of Address Slots} \times \text{Number of Bits per Entry}
\]

\[
\begin{align*}
\text{Number of Entries} &= 4096 \\
\text{Number of Address Slots} &= 4 \\
\text{Number of Bits per Entry} &= 2 \\
\end{align*}
\]

\[
\text{Total Bits} = 4096 \times 4 \times 2 = 32768
\]
59 – Correlated Branch Predictor Performance

The best we can do is compare program performance, in this case benchmarks.

![Comparison of two-bit predictors.](image)

FIGURE 4.21 Comparison of two-bit predictors.

60 – What we Learned

We observed about branch prediction buffer techniques:

1. Using too responsive a predictor is dangerous.
   Solution — Use more prediction information to “remember” the history.

2. The branch behavior between nearby branches can be correlated.
   Solution — Try incorporating correlation information into the prediction.

3. Hash collisions among branches can screw up branch prediction.
   Solution — Use a sufficiently large branch prediction table.

61 – Branch Target Buffers

We can get a bigger win out of branch prediction if we save the target address of the branch too. Note that hash collisions cannot be tolerated in this system, so the branch address must be stored.

![A branch-target buffer.](image)

FIGURE 4.22 A branch-target buffer.

62 – Pipelining and Branch Target Buffers

On average, a high hit rate and prediction probability can reduce stalls.

![The steps involved in handling an instruction with a branch-target buffer.](image)

FIGURE 4.23 The steps involved in handling an instruction with a branch-target buffer.
63 – In Class Exercise

Assume that:
1. the branch taken frequency is 60%,
2. the prediction accuracy is 90% and
3. the hit rate in the buffer is 95%.

What is the average branch penalty given that the following penalties:

<table>
<thead>
<tr>
<th>Buffer Hit</th>
<th>Prediction</th>
<th>Actual</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Taken</td>
<td>Taken</td>
<td>0 Cycles</td>
</tr>
<tr>
<td>Yes</td>
<td>Taken</td>
<td>Not Taken</td>
<td>2 Cycles</td>
</tr>
<tr>
<td>No</td>
<td>Taken</td>
<td></td>
<td>2 Cycles</td>
</tr>
</tbody>
</table>

---

64 – About Branch Target Buffers

Suppose that the branch target buffer predicted both branches which were taken and those which are not taken. Then the set of penalties in cycles might look like:

<table>
<thead>
<tr>
<th>Buffer Hit?</th>
<th>Actual Branch Prediction</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Yes</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>No</td>
<td>T</td>
<td>NT</td>
</tr>
<tr>
<td>No</td>
<td>NT</td>
<td>NT</td>
</tr>
</tbody>
</table>

A predict NT branch has the same penalty as an unpredicted branch, so it can be removed from the table.

---

65 – Solution

Assuming that only predict taken branches populate the table we get:

Branch Penalty = \[2 \text{cycles} \times \text{Buffer Hit Rate} \times \left(1 - \text{Prediction Accuracy}\right) + \left(2 \text{cycles} \times \text{Buffer Hit Rate} \times \left(1 - 0.95 \times 0.9\right)\right)\] + \[0.190 + 0.090/\text{cycles} = 0.28\text{cycles}\]

---

66 – Stack Based Predictors

Evidence shows that run time computed destinations for branches are primarily induced by function calls, so using a stack heuristic for storing the addresses can give good performance:

---

FIGURE 4.25 Prediction accuracy for a return address buffer operated as a stack.
**67 – Multiple Issue and ILP**

We now will consider allowing concurrent issue of multiple instructions via:

1. Superscalar processors — Varying number of instructions per cycle (Compiler and/or Tomasulo)
2. Very Long Instruction Word (VLIW) processors — Fixed number of instructions per cycle (Compiler Scheduled) Explicitly Parallel Instruction Computing (EPIC) (Intel-HP Merced)

---

**68 – Superscalar Processors**

You can think of a superscalar machine as maintaining multiple pipelines with a common cycle time. Below the integer and FP pipelines are run concurrently.

<table>
<thead>
<tr>
<th>Op Type</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>WB</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>EX</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
</tr>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ID</td>
<td>ID</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
</tr>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ID</td>
<td>ID</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
</tr>
<tr>
<td>Integer</td>
<td>FP</td>
<td>FP</td>
<td>FP</td>
<td>FP</td>
<td>FP</td>
<td>FP</td>
<td>FP</td>
<td>FP</td>
</tr>
</tbody>
</table>

---

**70 – In Class Exercise**

Assume that the following FP latencies are in place:

<table>
<thead>
<tr>
<th>Producer of Result</th>
<th>User of Result</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU Op</td>
<td>FP ALU Op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU Op</td>
<td>Store Double</td>
<td>2</td>
</tr>
<tr>
<td>Load Double</td>
<td>FP ALU Op</td>
<td>1</td>
</tr>
<tr>
<td>Load Double</td>
<td>Store Double</td>
<td>0</td>
</tr>
</tbody>
</table>
71 - In Class Exercise

Recall the loop we know and love:

\[ \text{R0 is initially the high order memory location of } x \]

\[ \text{F2 is } s \]

\[ \text{F0, F1, F2, F4, F5 are } \pm 2 \text{ stalls} \]

\[ \text{R0 = 0} \]

\[ \text{store } x[0], & X[0] = 0 \]

How fast is an efficient superscalar scheduling be (hint unroll it)?

---

71 - Solution

The Unrolled Loop Can Be Scheduled:

<table>
<thead>
<tr>
<th>Integer FU</th>
<th>FP FU</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: LD F0, 0(R1)</td>
<td>ADDD F4, F0, F2</td>
<td>1</td>
</tr>
<tr>
<td>LD F6, -8(R1)</td>
<td>ADDD F6, F6, F2</td>
<td>2</td>
</tr>
<tr>
<td>LD F10, -16(R1)</td>
<td>ADDD F4, F4, F2</td>
<td>3</td>
</tr>
<tr>
<td>LD F14, -24(R1)</td>
<td>ADDD F12, F10, F2</td>
<td>4</td>
</tr>
<tr>
<td>LD F18, -32(R1)</td>
<td>ADDD F16, F14, F2</td>
<td>5</td>
</tr>
<tr>
<td>SD 0(R1), F4</td>
<td>ADDD F20, F18, F2</td>
<td>6</td>
</tr>
<tr>
<td>SD -8(R1), F8</td>
<td>ADDD F20, F18, F2</td>
<td>7</td>
</tr>
<tr>
<td>SD -16(R1), F12</td>
<td>ADDD F20, F18, F2</td>
<td>8</td>
</tr>
<tr>
<td>SUBI R1, R1, #40</td>
<td>ADDD F20, F18, F2</td>
<td>9</td>
</tr>
<tr>
<td>SD -24(R1), F16</td>
<td>ADDD F20, F18, F2</td>
<td>10</td>
</tr>
<tr>
<td>BNEZ R1, LOOP</td>
<td>ADDD F20, F18, F2</td>
<td>11</td>
</tr>
<tr>
<td>SD -32(R1), F20</td>
<td>ADDD F20, F18, F2</td>
<td>12</td>
</tr>
</tbody>
</table>

Which corresponds to 5 unrollings, and runs in 2.4 cycles per element as opposed to 3.5 cycles per element of unrolling alone.

---

73 - Multiple Issue with Dynamic Scheduling

Suppose we want to adapt Tomasulo's algorithm to issue cycle such that two instructions can be issued simultaneously (say an FP and an integer instruction).

This imposes the restrictions:

1. Instructions should be issued to reservations in order (for reduced complexity).
2. Instructions with dependencies should not be issued in the same clock cycle.

Some tools to help are:

1. The issue cycle can be pipelined to allow multiple issue and
2. Only worry about FP loads and moves from GP to FP registers inducing dependences.

---

74 - A Hybrid Static/Dynamic Scheduling Policy

We can use a decoupled architecture which has:

- Dynamic Scheduling — Instead of using reservation stations use a queue of FU's waiting for the results of a load or store.
- Static Scheduling — Use the compiler to analyze dependences between other instructions.

Note that the dynamic scheduling of loads and stores requires hazard detection.
75 – Dynamic Scheduling Limitations

1. Sensitivity to instruction mix — In the integer/FP split we need a 50/50 mix of instruction types to get CPI = 0.5.
2. Complexity increases with the number of multiple issued instructions.
3. Hazard detection burden falls on hardware system.

76 – VLIW

Very Long Instruction Word (VLIW) machines are based on the following assumptions:

1. Dependence testing in hardware is complex, and scheduling might be better handled by the compiler.
2. As per RISC, only a few instructions are used, and they can be encoded in a small space.
3. Most operations use small operands, so we don’t need a large operand encoding space.

Let’s consider an example expected to come to market by 2000, the HP/Intel IA-64 Merced.
79 – Branching limits Parallelism

Branch instructions in traditional architectures enforce sequential execution.

80 – Predication

Predication “tags” both the body of the if and else and speculatively runs them in parallel.

81 – Predication and Parallelism

Predication can enhance parallelism.

82 – Predication in Practice

Real World Examples!

1. Expanded ISA of Alpha, MIPS, Power PC, SPARC have conditional move operations (i.e. annull only loads and stores).

2. HP PA-RISC and IA-64 can annull any instruction.

Predication has limits:

1. “Anulled” operations still take a clock cycle.

2. Late evaluation of conditions induces stalls.

3. Complex conditions necessarily have late evaluations.

Note: Predication increases basic block length, and helps with static scheduling.
83 – Predication in the IA-64

How Predication Works

1. The branch has two possible outcomes.
2. The compiler analyzes a predicate register to each following instruction, according to its path.
3. All instructions along this path point to predicate register P1.
4. All instructions along the path point to predicate register P2.
5. CPU begins executing instructions from both paths.
6. CPU can execute instructions from both paths in parallel because they have no mutual dependencies.
7. When CPU knows the compare outcome, it discards results from invalid path.

128-bit long instruction words

Predication replaces branch prediction by allowing the CPU to execute all possible branch paths in parallel.

84 – Trace Scheduling

Sometimes the branching is complex, so we apply static scheduling to help.

The Compiler (with perhaps a profiler):

1. Trace Selection — Find a trace (a sequence of basic blocks executed with high probability).
2. Trace Compaction — Makes the fast case correct and infrequent case correct by:
   (a) Place the trace into a small number of VLIW instructions.
   (b) The compiler generates fixup instructions for wrong predictions wrong.

Downside — The compiler technology is hard to get correct.

85 – Memory Access Limits Parallelism

Load operations in particular can induce muticycle pipeline delays.

86 – Speculative Loading

Speculative loading is a combined hardware/software approach (in IA-64).
87 – IA-32 Features

The IA-32 is a 32 bit CISC architecture on the surface (Pentium Pro). The Pentium Pro (according to Patterson in Fall '96 lecture notes):

1. Uses a sort of Scoreboard like algorithm for scheduling.
2. Does NOT pipeline 80x86 instructions.
3. Decodes them using a 72 bit microcode (which is similar to DLX).
4. Dispatches microinstructions to a reorder buffer and reservation stations.
5. Uses 1 cycle to compute the length and and 2 to generate the microinstructions.
6. 12-14 clocks in the pipeline
7. Many instructions require 1 to 4 microinstructions.
8. Complex instructions are done via microprograms (longer sequences of microinstructions).

88 – IA-64 and Legacy IA-32 Codes

The IA-64 should run IA-32 code.

Furthermore, mixing of IA-64 and IA-32 modes should be available.

The patent application gives a variety of mechanisms for doing this including:

To support the mixed mode programming, Intel has patented the interface instructions for:

1. Instructions for transferring register contents between the IA-64 and 80x86 mode.
2. JMPX/JMPX86 — Give control the the IA-64/80x86 processor.

89 – VLIW vs. SuperScalar Architecture

Hardware (Tomasulo) vs. Software (VLIW)

Speculation:
1. HW better determines address conflicts
2. HW does better branch prediction
3. HW maintains precise exception model (debugging?)
4. HW avoids book keeping instructions.
5. Both may experience poor performance if code compiled for older versions is run on newer machines.
6. SW speculation is much easier for HW designer (but tough for compiler writers).
7. HW dependencies may stop instruction issue.

EVRET — Event return in IA-64 mode. An event is a machine check, interrupt or exception.
IRET — 80x86 event return is extended in functionality so that it can return control to the IA-64.
References

Compilers, principles, techniques, and tools.