Instruction Level Parallelism and Its Dynamic Exploitation

Instruction Level Parallelism achived from overlapping executions of multiple instructions. Increasing amount of parallelism requires reducing the effect of hazards and using compilers.

**Performance in pipeline**

\[
\text{CPI} = \text{Ideal CPI} + \text{Structural hazards stalls} + \text{RAW stalls} + \text{WAR stalls} + \text{WAW stalls} + \text{Control stalls}
\]

Different techniques to reduce R.H.S:

- Control stalls by Loop unrolling and speculation and dynamic branch prediction
- RAW stalls by dynamic scheduling with scoreboarding, basic pipeline scheduling, speculation

N. Mekhiel
Different techniques to reduce R.H.S:

- Ideal CPI by issuing multiple instructions, compiler dependency analysis, software pipelining, and trace scheduling
- WAR and WAW stalls by dynamic scheduling and register renaming and speculation

**Concept of ILP**
Amount of parallelism in a straight line sequence of code.
Limitations: with a branch frequency of 15%, means on average there will be only 6-7 instructions between any pair of branches. This means that maximum amount of overlap is much less than 6.
We must find parallelism outside each basic block (among blocks).

N. Mekhiel
Finding Parallelism Among Iterations of a Loop

Example:

```c
for (i=1; i<=1000; i++)
    X[i] = X[i] + Y[i];
```

There is a loop parallelism (1000) that we need to convert it to ILP.

Method: Loop Unrolling using compiler or hardware techniques.

Other Method: Use Vector Processing.
Tradeoffs: Increasing size of code and using more registers.

Key to improvements, resolve dependency between instructions.
N. Mekhiel
Dependences

Two instructions that are independent are parallel and could be executed in parallel.

Types of Dependences:

1-Data Dependences

1- Instruction i produces results that is used by j
2- Instruction j dependens on k and k depends on i (indirect dependency)

Example:

1 Loop: LD F0, 0(R1) ; F0= element
2 ADDD F4, F0, F2 ; Add scalar to the element
3 SD 0(R1), F4 ; store new value to element

Data dependency between I1, I2 (F0)
and between I2,I3 (F4)

4 SUBI R1,R1, #8 ; Decrement pointer
5 BNE R1, R2, Loop ; branch if R1 != R2

Data dependency between 4, 5 (R1)

N. Mekhiel
The presence of dependences indicates a potential for a hazard, but actual hazard and length of stall is a property of pipeline.

**Example of Data dependence with dependent chain:**

1-Loop: \[ \text{LD } F0, 0(R1) \]
2- \[ \text{ADDD } F4, F0,F2 \]
3- \[ \text{SD } 0(R2), F4 \]
4- \[ \text{SUBI } R1, R1, #8 \]
5- \[ \text{LD } F6, 0(R1) \]
6- \[ \text{ADDD } F8, F6, F2 \]
7- \[ \text{SD } 0(R1), F8 \]
8- \[ \text{SUBI } R1, R1, #8 \]

Data dependency between Inst4, Inst5, Inst7, Inst8 for R1

Compiler removes these dependences by computing intermediate values of R1 and adjusting the offset of LD, SD and decrementing R1 removing SUBI. \[ R1 = R1 -8 \text{ (LD } F6, -8(R1),..) \]

N. Mekhiel
Name Dependences

Two types:
1-WAR hazard (antidependence)
When instruction j writes to register or memory before instruction i reads it
2-WAW hazard (output dependence)
When j writes to same register or memory before i writes to it.
There is no data being transferred between instruction i, j. We can use register renaming to resolve this.
It is difficult to detect same name for memory access. For example 0(R2) and 20(R3) might reference the same memory location.

N. Mekhiel
Example: With loop unrolling, no scheduling, indicate both data and name dependences. Show how renaming reduces name dependences.

1–LD F0, 0(R1)  
2–ADDD F4, F0, F2  
3–SD 0(R1), F4  
4–LD F0, −8(R1)  
5–ADDD F4, F0, F2  
6–SD −8(R1), F4

N. Mekhiel
1- Loop: LD F0, 0(R1)
   2- ADDD F4, F0, F2
   3- SD 0(R1), F4
   4- LD F0, -8(R1)
   5- ADDD F4, F0, F2
   6- SD -8(R1), F4
   7- LD F0, -16(R1)
   8- ADDD F4, F0, F2
   9- SD -16(R1), F4
  10- LD F0, -24(R1)
  11- ADDD F4, F0, F2
  12- SD -24(R1), F4
  13- SUBI R1, R1, #32
  14- BNEZ R1, Loop

N. Mekhiel
With Register Renaming

1– Loop: LD F0, 0(R1)

2– ADDD F4, F0, F2

3– SD 0(R1), F4

4– LD F6, −8(R1); rename F0 −−> F6

5– ADDD F8, F6, F2; rename F4 −−> F8

6– SD −8(R1), F8;

7– LD F10, −16(R1) rename F0 −−>F10

8– ADDD F12, F10, F2; rename F4 −−>F12

9– SD −16(R1), F12

10– LD F14, −24(R1); rename F0−−> F14

11– ADDD F16, F14, F2; rename F4 −−> F16

12– SD −24(R1), F16

13– SUBI R1, R1, #32

14– BNEZ R1, Loop

N. Mekhiel
Control Dependences
Control dependences: It determines if a set of instructions should be executed or not. Control dependences must be preserved.

Example:

1- if P1 {Si}
   ;

Instructions on Si should only be executed if P1 is true.

We cannot move an instruction that is control dependent branch to before branch (cannot move instructions before if).

2- X;
   if P2 {S2;}

Instruction that is not control dependent cannot be after branch.
Cannot move statement before (X;) if to into the branch.

N. Mekhiel
Example: Indicate control dependences for loop unrolling with overhead

SUBI, BEQZ

Loop:  
LD    F0, 0(R1)  
ADDD  F4, F0, F2  
SD    0(R1), F4  
SUBI  R1, R1, #8  
BEQZ  R1, EXIT  

LD    F6, 0(R1)  
ADDD  F8, F6, F2  
SD    0(R1), F8  
SUBI  R1, R1, #8  
BEQZ  R1, EXIT  

LD    F10, 0(R1)  
ADDD  F12, F10, F2  
SD    0(R1), F12  
SUBI  R1, R1, #8  
BEQZ  R1, EXIT  

LD    F14, 0(R1)  
ADDD  F16, F14, F2  
SD    0(R1), F16  
SUBI  R1, R1, #8  
BNEZ  R1, Loop  

Control Dependency on BEQZ

If R1 = 48, We can remove 3 intermediate SUBI, BNEZ, and remove Control dependences.

N. Mekhiel
Preserving Control Dependency By:
1-In order instruction execution
2- Detection of Control hazards and stall

Preserving control dependences assures program correctness in two ways:

1-Preserve exception behavior

Example:

    BEQZ R2, L1
    LW R1, 0(R2)

Assume that control dependence is not preserved, and we move LW before branch.

LW may cause an exception (memory violation), if branch is taken, the above exception will never occur.

N. Mekhiel
2-Preserve data flow:

branches make data flow dynamic, hard to predict in static code and cannot be guaranteed with data dependency.

Example:

    ADD R1, R2, R3
    BEQZ R4, L
    SUB R1, R5, R6
    L:   OR R7, R1, R8

If branch is taken $R1 = R2 + R3$,
    IF not $R1 = R5 - R6$

N. Mekhiel
Dynamic Scheduling To Deal with Data Hazards

Without dynamic scheduling, we used compiler to minimize hazards using static scheduling.

dynamic scheduling uses Hardware to rearrange instructions to reduce stalls.

Advantages of Dynamic Scheduling:

- Deals with dependences which are only known at run time.

- Simplifies Compiler

- Allows code compiled for one pipeline to run on a different pipeline.

Dynamic Scheduling allows out of order Instruction Completion making Precise Exceptions difficult.

N. Mekhiel
Example:

```
DIVD   F0, F2, F4
ADDD   F10, F0, F8 ;RAW dependency on F0
SUBD   F12, F8, F14
```

Normal pipeline will stall pipeline waiting for DIVD to complete.

SUBD is not data dependent on anything.

With dynamic scheduling SUBD can complete early out of order.

N. Mekhiel
Concept of Dynamic Scheduling Using Scoreboarding

Pipeline with Scoreboarding

Pipeline: IF; Queue, Decode(Issue, Read Operand), Execute; Write Back

N. Mekhiel
Dynamic Scheduling:

- Instruction queue allows multiple instructions to be available for issuing (if one instruction stall, others are available).

- Decode stage is split in two stages:-
  - 1-Issue: Decode instruction and check for structural hazards
  - 2-Read operand: Wait until no data hazards (RAW)

- Execute stage might take multiple clock cycles

- Write Back to write results

- Scoreboard: monitors hazards, dependency, available FUs, operands, and change pipeline to execute instructions as early as possible.

N. Mekhiel
Scoreboarding creates a new hazard:

It could create a WAR hazard that did not exist in simple DLX pipeline.

Example:

```
DIVID F0, F2, F4
ADDD F10, F0, F8; RAW for F0
SUBD F8, F8, F14; WAR for F8
```

If we let SUBD completes early before ADDD, we have WAR hazard. Scoreboard must detect this hazard.

If destination register for SUBD = F10, we have WAW. Scoreboard must also check for WAW hazards.

N. Mekhiel
Multiple Function Units to allow execution of multiple instructions.

Scoreboarding with Multiple Function Units

Latency of MULT = 10 cycles; DIV = 40 cycles; ADDD = 2 cycles, Int = 0 cycle.
Four Stages of Scoreboard Control:

- **1-ISSUE:** Decode instruction and check for structural hazards
  - a-Functional Unit is free
  - No active instruction has the same destination (WAW). If not, stall issuing any instruction (in order issuing), wait until a,b are okay.

- **2- Read Operands:**
  Wait until no data hazards, then read operands (RAW)
  - a-no active instruction is going to write to operand.
  - no function unit is currently writing to this operand

If a,b okay instruction may be sent to execute OUT OF ORDER.

N. Mekhiel
Stages of Scoreboard

- **3-Execution**: Operate on operands and might take multiple cycles. When result is ready, it notify scoreboard for completion.

- **4- Write results:**
  - a-Scoreboard check for WAR (DEST is not used as source for other pending instructions)
    
    Example:
    ```
    DIVD F0, F2, F4;
    ADDD F10, F0, F8;
    SUBD F8, F8, F14;
    Instruction SUBD will not write to F8 until ADDD reads F8.
    ```
  - b-Stall until all pending instructions read their sources.

N. Mekhiel
# Parts of Scoreboard

Three Parts of Scoreboard

## 1–Instruction Status in Pipeline

<table>
<thead>
<tr>
<th>Issue</th>
<th>Read Operands</th>
<th>Execute Complete</th>
<th>Write result</th>
</tr>
</thead>
</table>

## 2–Function Units Status

<table>
<thead>
<tr>
<th>Busy</th>
<th>OP</th>
<th>Fi</th>
<th>Fj</th>
<th>Fk</th>
<th>Qj</th>
<th>Qk</th>
<th>Rj</th>
<th>Rk</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU is busy or not</td>
<td>operation in FU</td>
<td>dest. Reg</td>
<td>source Reg1</td>
<td>source Reg2</td>
<td>FU that produces Fj</td>
<td>FU that produces Fk</td>
<td>Flag if Fj ready</td>
<td>Flag if Fk ready</td>
</tr>
</tbody>
</table>

- Issue if (FU, Fi) ready
- Read if (Rj, Rk) not used by any FU (Qj, Qk)
- Execute
- Write result if (Fj, Fk) not used by any FU (Qj, Qk)

## 3–Register result status

<table>
<thead>
<tr>
<th>Reg</th>
<th>F0</th>
<th>F8</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU</td>
<td>DIV</td>
<td>ADD</td>
</tr>
</tbody>
</table>

N. Mekhiel
Add Latency = 2 cycles
Mult Latency = 10 cycles
Div Latency = 40 cycles
Int Latency = 0 cycles

N. Mekhiel
## CYCLE 5

<table>
<thead>
<tr>
<th></th>
<th>Issue</th>
<th>Read</th>
<th>Complete</th>
<th>Write result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F6, 34(R2)</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>LD</td>
<td>F2, 45(R3)</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multd</td>
<td>F0, F2, F4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subd</td>
<td>F8, F6, F2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divd</td>
<td>F10, F0, F6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addd</td>
<td>F6, F8, F2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Name

<table>
<thead>
<tr>
<th>busy</th>
<th>op</th>
<th>Fi</th>
<th>Fj</th>
<th>Fk</th>
<th>Qj</th>
<th>Qk</th>
<th>Rj</th>
<th>Rk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>LD</td>
<td>F2</td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Mult1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Div</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Reg

<table>
<thead>
<tr>
<th></th>
<th>F0</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fu</td>
<td>Int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(N. Mekhie)
### CYCLE 8

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Read</th>
<th>Complete</th>
<th>Write result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6, 34(R2)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>LD F2, 45(R3)</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Multd F0, F2, F4</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subd F8, F6, F2</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divd F10, F0, F6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addd F6, F8, F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Name

<table>
<thead>
<tr>
<th>busy</th>
<th>op</th>
<th>Fi</th>
<th>Fj</th>
<th>Fk</th>
<th>Qj</th>
<th>Qk</th>
<th>Rj</th>
<th>Rk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>LD</td>
<td>F2</td>
<td></td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult1</td>
<td>Mult</td>
<td>F0</td>
<td>F2</td>
<td>F4</td>
<td>Int (F2)</td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Mult2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Sub</td>
<td>F8</td>
<td>F6</td>
<td>F2</td>
<td>Int (F2)</td>
<td>Y</td>
<td>N (F2)</td>
<td></td>
</tr>
<tr>
<td>Div</td>
<td>Div</td>
<td>F10</td>
<td>F0</td>
<td>F6</td>
<td>Mult</td>
<td>N (F0)</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

### Reg

<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>Mult</td>
<td>F2</td>
<td>F4</td>
<td>F6</td>
<td>F8</td>
<td>F10</td>
</tr>
</tbody>
</table>

### Fu

<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>Mult</td>
<td>Add</td>
<td>Div</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(N. Mekhiel)
Reducing Branch Penalties with Dynamic Hardware Prediction

Motivations :-

- Reduce control dependences and extract more ILP.
- Reduce effect of branches in multiple issuing (branch comes n times faster).
- Branch penalty has very dramatic effect in fast machines (Amdahl’s law).

Background:
Static prediction schemes used to deal with branches (taken/not taken) does not depend on dynamic behavior of branch. 
Delayed branch scheme uses compiler to schedule useful instruction (static).
We need more accurate branch prediction and has to follow dynamically the branch behavior.

Concept: Use hardware to dynamically predict branch outcome, early.
N. Mekhiel
Branch Prediction Buffer

Uses buffer (memory) to store recent branch behavior (1 bit/2 bit) for prediction. The buffer is indexed by the lower portion of branch instruction address. It uses Multiplexer to select BTA or PC depending on (value of 1 bit/2 bit) prediction attached to each branch address.

N. Mekhiel
One bit prediction scheme has performance shortcoming as it is likely to predict incorrectly twice rather than once.

Example:

L1: ----- 
     ----- 
L2: ---- 
    br L2 
     ----- 
    br L1

assume a loop and branch always taken, then when not taken, we predict wrong (1 time), and buffer is updated with 1 bit=not taken. Next time, we enter the loop, the branch should be taken but we predict not taken, wrong again for the 2nd time.

N. Mekhiel
2 Bit Prediction Scheme
Prediction must miss twice before it changes the prediction.

Accuracy of branch prediction scheme
Using SPEC89, with prediction buffer of 4096 (12 bits idex), accuracy= 99% to 82%. Performance depends on:- branch frequency, prediction accuracy and misprediction penalties (cycles).
N. Mekhiel
Improving Branch Prediction with Two Level Predictors

It uses extra 2 bits for global history to choose among 4 predictors.

Example: If branch prediction buffer has 8 Kbits, using (2,2), find number of branch enteries.
Number of enteries $= 8K \div (4 \times 2bits) = 1$ K.
N. Mekhiel
Reducing Branch Penalty Using Branch Target Buffers "BTB"

Motivations: even with very good prediction, if misprediction penalty is high, the performance of branches will be low.

Concept: reduce branch penalty by predicting address of next instruction after a branch. This reduce branch penalty to zero.

It uses a branch target buffer to store PC of instruction to fetch.

IF1 ID1 EX1 ......; Branch instruction and BTB gives address

IF2 ID2 EX2 ; IF2 in time
If $PC = \text{Match}$, then $PC = \text{predicted PC}$ from branch target buffer. If $PC = \text{No match}$, then instruction is not predicted to be a branch and use $PC$.

**Improving Performance of BTB**

Storing Target Instruction instead of target address, when BTB hit, get instructions from BTB not cache (can use larger buffer for multiple instruction storage), It allows branch folding to obtain zero cycle branches)

N. Mekhiel
Techniques to Detect Loop Level Parallelism
Must analyze source code for data dependences in the Loop or across the iterations of Loop. **Data dependency detected if operand is written at some point and read at later point**

Examples of data dependences:
Example1: Find if there is dependences in loop body and between different iterations of:
\[
\text{for}(i=1; \ i<=1000; \ i++) \\
X[i]=X[i] \ +5;
\]
There is dependency within the loop $X[i], \ X[i]$
There is no dependency between iterations
Example2: find dependences in the following:
\[
\text{for}(i=1; \ i<=1000; \ i=i+1)\
\begin{align*}
A[i+1] &= A[i] \ + C[i]; \ S1 \\
B[i+1] &= B[i] \ + A[i+1]; \ S2
\end{align*}
\]
-There is dependency within Loop between $S1, S2$
using $A[i+1]$
-There is dependence between iteration $i$ and $i+1$
$S1$ writes to $A[i+1]$ and read at next iteration in $S1$ as $A[i]$
-the same in $S2$ for $B[i+1]$
Example 3: Find dependences in the following:

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

Iteration i:
```
A[i] = A[i] + B[i];  \text{ S1}
B[i+1] = C[i] + D[i];  \text{ S2}
```

Iteration i+1:
```
A[i+1] = A[i+1] + B[i+1];  \text{ S1}
B[i+2] = C[i+1] + D[i+1];  \text{ S2}
```

B[i] makes S1 depend on S2, but S2 does not depend on S1, there is no cyclic dependency and loops could be made parallel using LOOP TRANSFORMATION AS:
```
for(i=1; i<=99; i=i+1){
    B[i+1] = C[i] + D[i];  \text{ S2}
    A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] = D[100];
```
ILP with Loop Unrolling

For ILP to work, we must find sequence of unrelated instructions, avoid pipeline stalls, separate dependent instruction from source by a distance equal to **pipeline latency**

**FP Pipeline Latency**

<table>
<thead>
<tr>
<th>Inst. Producing result</th>
<th>Inst. Using it</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU OP</td>
<td>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>

**Examples of Loop Unrolling**

```plaintext
for(i=1; i<=1000; i++)
    x[i] = x[i] + s;
assume  M[R1]=x[1000], F2=s
Loop: LD F0, 0{R1) ; F1= x[i]
      ADDD F4, F0, F2 ; F4= x[i] + s
      SD 0(R1), F4     ; x[i]=F4
      subi R1, R1, #8 ; i-1
      BNEZ R1, Loop
```
Performance without unrolling loop:

\[
\text{for}(i=1; \ i\leq1000; \ i++)
\]
\[
\quad x[i] = x[i] + s;
\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: LD F0, 0{R1);</td>
<td>1</td>
</tr>
<tr>
<td>STALL 1 CYCLE</td>
<td></td>
</tr>
<tr>
<td>ADDD F4, F0, F2;</td>
<td>3</td>
</tr>
<tr>
<td>STALL 2 CYCLES</td>
<td></td>
</tr>
<tr>
<td>SD 0(R1), F4;</td>
<td>6</td>
</tr>
<tr>
<td>SUBI R1, R1, #8;</td>
<td>7</td>
</tr>
<tr>
<td>STALL 1 CCLE</td>
<td>8</td>
</tr>
<tr>
<td>BNEZ R1, Loop;</td>
<td>9</td>
</tr>
<tr>
<td>STALL 1 CYCLE</td>
<td>10</td>
</tr>
</tbody>
</table>

NUMBER OF CLOCKS = 10 CYCLES

Using Scheduling:

Loop: LD F0, 0(R1) | 1 |
| SUBI R1, R1, #8 | 2 |
| ADDD F4, F0, F2 | 3 |
| BNEZ R1, Loop | 4 |
| STALL 1 CYCLE | 5 |
| SD =8(R0), F4 | 6 |

Number of Cycles = 6 (more than 50% gain)
Loop Unrolling 4 times

<table>
<thead>
<tr>
<th>Instruction</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0, 0{R1};</td>
<td>1</td>
</tr>
<tr>
<td>STALL 1 CYCLE</td>
<td></td>
</tr>
<tr>
<td>ADDD F4, F0, F2;</td>
<td>3</td>
</tr>
<tr>
<td>STALL 2 CYCLES</td>
<td></td>
</tr>
<tr>
<td>SD 0(R1), F4;</td>
<td>6</td>
</tr>
</tbody>
</table>

1st

LD F6 -8(R1)
ADDD F8, F6, F2
Stall 2 cycles
SD -8(R1), F8

12

LD F10 -16(R1)
ADDD F12, F10, F2
Stall 2 cycles
SD -16(R1), F12

18

3rd
Unrolling the Loop for 4th tie

LD F14 -24(R1)
ADDD F16, F14, F2
Stall 2 cycles
SD -24(R1), F16
SUBI R1, R1, #32
STALL 1 CYCLE
BNEZ R1, Loop ;
STALL 1 CYCLE

Total= 27 for 4 loops (6.75 per
Loop Unrolling + Scheduling

Loop: LD F0, 0(R1); F0 = \( x[1000-4i] \)
LD F6, -8(R0)
LD F10, -16(R1)
LD F14, -24(R1)
ADDD F4, F0, F2; F4 = \( x[1000-4i] + S \)
ADDD F8, F6, F2
ADDD F12, F10, F2
ADDD F16, F14, F2
SD 0(R1), F4; \( x[1000-4i]=F4 \)
SD -8(R1), F8
SUBI R1, R1, #32
SD -16(R1), F12
BNEZ R1, Loop
SD +8(R1), F16; \( x[1000-4i-3]=F16 \)

Total #cycles=4x3+2=14, or 14/4=3.5 cycles
9/3.5 almost 300% gain in performance
**Multiple Issue**
To improve ideal CPI, make it less than 1 (IPC instructions per clock cycle).
Two types:-

- **Superscalar**

- **VLIW** (very long instruction word)

**1-Superscalar**

- Issue variable number of instructions per clock.

- Instructions must be independent, no more than one memory reference per clock

- Variable instruction issuing, and dynamic issuing

**Simple DLX version:** Integer and FP Operations could be issued simultaneously.

- 1-Fetch and Decode 2 instructions 64 bits per cycle

- 2-Hardware will issue 2nd instruction (FP) if 1st instruction can issue (dynamically).
**DLX Superscalar**

- 3-Need to pipelined FP Units or use multiple FP Function Units (Issuing of FP Inst 1/cycle).

- 4-Using different Register Sets for Integer and FP.

- 5-Hazards must be detected if having Int Instruction is a FP load, FP Inst. has dependency on load Inst (RAW).

- 6-Problem with DLX Load latency of one clock cycle= cost 3 instructions (1+2) and branch delay latency of 1 cycle = 3 instructions.

Find Performance of DLX Superscalar for the following:
Loop: LD F0 0(R1)
  ADDD F4, F0, F2
  SD 0(R1), F4
  SUBI R1, R1, #8
  BNEZ R1, Loop
Assume Loop unrolled 5 times.
Superscalar gain $= 17/12 = 1.41$

Performance of superscalar $= 12 \div 5 = 2.4$ cycles per loop.
Performance gain $= 17 \div 12 = 1.41$

Superscalar could use dynamically scheduled instructions with scoreboard (Tomasulo’s algorithm) to improve ILP.
VLIW Approach

- Reduces hardware needed for multiple issue superscalar
  Superscalar hardware is needed to examine opcode of multiple instructions, and registers to determine if it can issue them.

- VLIW use multiple independent functional Units and one long instruction with the help of compiler to form it (static scheduling).

- VLIW instruction might have 2 Int operations + 2 FP operations + 2 memory operations and a branch. Instruction length could be 112 or 168 bits.

Example
Assume 2 mem operations, 2 FP, 1 Int 1 branch in VLIW cycle. Using loop unrolling for 7 times, find the performance improvement for VLIW
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>loop LD F0, 0(R1)</td>
<td>LD F6, −8(R1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F10, −16(R1)</td>
<td>LD F14, −24(R1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F18, −32(R1)</td>
<td>LD F22, −40(R1)</td>
<td>ADD F4, F0, F2</td>
<td>ADD F8, F6, F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>LD F26, −48(R1)</td>
<td>ADD F12, F10, F2</td>
<td>ADD F16, F18, F2</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>ADD F12, F10, F2</td>
<td>ADD F20, F18, F2</td>
<td>ADD F24, F22, F2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>SD 0(R1), F4</td>
<td>SD −8(R1), F8</td>
<td>ADD F28, F26, F2</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>SD −16(R1), F12</td>
<td>SD −24(R1), F16</td>
<td></td>
<td>SUBI R1, R1, # 56</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>SD −32(R1), F20</td>
<td>SD −40(R1), F24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD 8(R1), F28</td>
<td></td>
<td></td>
<td>BNZ R1, loop</td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

Performance gain=17/9=2

Performance of VLIW= 9 Cycles for 7 loop iterations=1.28 cycles/loop compared to 2.4 for superscalar or 3.5 with scheduling and loop unrolling.
Limitations of Multiple Issue Processors

- Inherent limitations of ILP in programs (very serious limitation). Not enough parallel operations.

- Complexity of hardware implementations, large increase in memory bandwidth, register bandwidth (multiple ports).

- Superscalar with dynamic scheduling complicates design

- VLIW increases code size, if instructions are not full (not enough ILP) causes a waste in instruction encoding. Cache miss in VLIW causes all function units to stall (all instructions in VLIW are synchronized).

- Binary code compatibility is a major problem for VLIW (comes from using different number of function Units, or instructions).
Pentium 4 Processor Microarchitecture

• 90 nm Process technology
• Execution trace cache
• 2X frequency execution core
• Hyper-Threading (SMT)
• New SSE3 Instructions (streaming SIMD Extension)
• Higher frequency with extended pipeline (3.4 GHz)
Overview of NETBURST (P4) microarchitecture

Execution trace cache
Out-of-order core
Rapid Execution
Store to load forwarding
Trace Cache:

- Instruction cache called execution trace cache
- Stores decoded instructions in form of uops
- UOPS could be accessed repeatedly like cache
- No Decoding, trace cache takes decoded uops from decoder
- Trace is assembled from multiple UOPS (up to 6) which are sequentially predicted from path of program including target of branches. In instruction cache, only branch instruction is delivered with delay but not from target.
- 3 UOPS per cycles
- When trace cache miss, fetch and decode from L2 cache
- Trace cache holds up to 12 K UOPS
Instructions that cannot be encoded in trace cache, use indirect CALLS and sequence them from Microcode ROM as indirect CALLS, and software prefetch.

Out of Order Core:

- Extract parallelism from code stream (UOPS)
- Schedule as many UOPS as possible for execution per each clock cycle
- Scheduler tracks input register operands when ready to execute and execution resources available
- Can dispatch 6 UOPS
- ALU can schedule on each 1/2 cycles
- Uses two dispatch ports for load and store
Execution Engine:

- Executes up to 6 UOPS per cycle:
  - 2 Int ALU
  - 1 Complex Int unit
  - load and store address generation units (AGU)
  - complex FP/media unit
  - FPmedia move unit
- INT ALU executes at double clock speed
- Execute 1 load and 1 store every clock from L1 (16 KB, 8 way, 64 B) Data cache
- Parallel DTLB and L1 to provide low latency from L1
Store to Load Forwarding:

• Could forward data to be stored to L1 directly to load using a store forwarding buffer (SFB)

• This is important because store is done at a later stage in a deep pipeline

• It uses feedback, MUX , SFB and control logic similar to data forwarding

Branch Prediction:

• Uses Static branch prediction (simple)

• prediction at decode time (early)

• If branch is backwards, predict it taken and if forward predict it not taken

• modify prediction for loop ending branch (with branch distance is less than a threshold)

• must flush pipeline if branch miss predicted
Memory System:

- Unified (I and D) L2 cache system size= 1 MB, 8 way, 128 B
- Use prefetch instructions for data to L2 and page table entries to DTLB
- Hardware prefetching by a predictor for stream of data

Hyper-Threading:

- Allows one physical processor to appear as two logical processors
- Two software threads can execute simultaneously eliminating context switching overhead penalty
- Changes in microarchitecture to support Hyper Threading:
  - Increase number of outstanding L1 load misses from 4 to 8
– Increase queue size between front end and allocation/rename logic

– Page table walk can occur at same time as memory access

– Page table walk that misses all caches and go to DRAM does not block other page table walk from being initiated

– Trace cache respond faster to stalling events and dedicate all its resources to the thread that is not stalled

– Uses extra bit in virtual tag for each logical processor to prevent conflicts in L1 cache when the two logical processors has a matched virtual tag (encourage true sharing but disallow false sharing).
SSE 3 Instructions:

- For Integer conversion
- Complex arithmetic
- Video encoding
- Graphics
- Thread synchronization

Pentium 4 Performance
Improvements due to speed and Hyper Threading of about 20% (limited why?).
EXAMPLES:

A-13 Scoreboard stage other than execute takes 1 cycle, 
MUL takes 3 cycles, ADD, SUB each take 1 cycle. Assume 
two function units and two multiply function units. Find 
processor pipeline stages in executing:

<table>
<thead>
<tr>
<th></th>
<th>Issue</th>
<th>Read</th>
<th>Execute</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL F0, F6, F4</td>
<td>1</td>
<td>2</td>
<td>2+3=5</td>
<td>6</td>
</tr>
<tr>
<td>SUB F8, F0, F2</td>
<td>2</td>
<td>7(MUL F0)</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>ADD F2, F10, F2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

CH4-5
List all dependency(true=RAW), (ou=WAW) and (anti=WAR) 
in the following:
for(i=2; i<100; i=i+1)
{

    a[i] = b[i] + a[i] ; S1
    c[i-1] = a[i] + d[i] ; S2
    a[i-1] = 2*b[i] ; S3
    b[i+1] = 2*b[i]; S4
Next loop

    a[i+1] = b[i+1] + a[i+1] ; S1
    c[i] = a[i+1] + d[i+1] ; S2
    a[i] = 2*b[i+1] ; S3
    b[i+2] = 2*b[i+1]; S4

within loop RAW S1, S2
out of loop RAW S4, S1 and S4, S3
out of loop WAW S1,S3
out of loop WAR S3 2ND LOOP, S1,S2 FIRST LOOP
Example 4-8

The following loop $Y[i] = axX[i] + Y[i]$

```
loop
    LD F0, 0(R1); F0=X[i]
    MUL F0, F0, F2; F0=a*X[i]
    LD F4, 0(R2); F4=Y[i]
    ADD F0, F0, F4; F0=a*X[i]+Y[i]
    SD 0(R2), F0; Y[i]=a*X[i]+Y[i]
SUBI R1, R1, #8; i+1 for X[i]
SUBI R2, R2, #8;
BNZ R1, loop
```

a-single-issue, unroll loop, and schedule it

Assume ALU-ALU=3, ALU-SD=2, LD ALU=1

```
loop
    LD F0, 0(R1)
    LD F6, -8(R1)
    LD F12, -16(R1)
    LD F18, -24(R1)
    MUL F0, F0, F2
    MUL F6, F6, F2
    MUL F12, F12, F2
    MUL F18, F18, F2
    LD F4, 0(R2)
    LD F8, -8(R2)
```
LD F10, -16(R2)
LD F14, -24(R2)
ADD F0,F0,F4
ADD F6,F6,F8
ADD F12,F12,F10
ADD F18,F18,F14
SUBI R1,R1,#8
SUBI R2,R2,#8
SD 8(R2), F0
SD 0(R2), F6
SD -8(R2), F12
BNZ R1, loop
SD -16(R2)
TIME/LDOP =23/4=5.75 CYCLES
assume dual-issue processor

loop LD F0, 0(R1)
  LD F6, -8(R1)
  LD F12, -16(R1)  MUL F0, F0, F2
  LD F18, -24(R1)  MUL F6,F6,F2
  LD F4, 0(R2)     MUL F12,F12,F2
  LD F8, -8(R2)    MUL F18,F18,F2
  LD F10, -16(R2)  ADD F0,F0,F4
  LD F14, -24(R2)  ADD F6,F6,F8
  SUBI R1,R1,#8    ADD F12,F12,F10
  SUBI R2,R2,#8    ADD F18,F18,F14
  SD 8(R2), F0
  SD 0(R2), F6
  SD -8(R2), F12
  BNZ R1, loop
  SD -16(R2)

total cycles=15/4=3.75
Example 4-9
a) find number of cycles

bar: LD F2,0(R1) 1
  stall
MUL F4,F2,F0 3
LD F6,0(R2) 4
  stall 2 cycles MUL to ADD
ADD F6,F4,F6 7
  STALL 2 CYCLES ALU-STORE
SD 0(R2),F6 10
ADDI R1,R1,#8 11
ADDI R2,R2,#8 12
SGTI R3,R1,#800 13
  STALL 1 CYCLE R3
BEQZ R3, bar 15
  STALL FOR BRANCH 1 CYCLE

TOTAL=16 CYCLES PER LOOP
b) single issue with loop unrolling 4 times + scheduling

bar

LD F2,0(R1)
LD F4,8(R1)
LD F6,16(R1)
LD F8,24(R1)
MUL F2,F2,F0
MUL F4,F4,F0
MUL F6,F6,F0
MUL F8,F8,F0
LD F10,0(R2)
LD F12,8(R2)
LD F14,16(R2)
LD F16,24(R2)
ADD F2,F2,F10
ADD F4,F4,F12
ADD F6,F6,F14
ADD F8,F8,F16
SD 0(R2),F2
SD 8(R2),F4
ADDI R1,R1,#8
ADDI R2,R2,#8
SGTI R3,R1,#800
SD 8(R2),F6
BEQZ R3, bar
SD 16(R2), F8

CYCLES=24/4=6 Cycles
c-using VLIW, unroll loop 4 times

<table>
<thead>
<tr>
<th>mem</th>
<th>mem</th>
<th>FP</th>
<th>FP</th>
<th>INT</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F2</td>
<td>LD F4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F6</td>
<td>LD F8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F10</td>
<td>LD F12</td>
<td>MLT F2</td>
<td>MLT F4</td>
<td></td>
</tr>
<tr>
<td>LD F14</td>
<td>LD F16</td>
<td>MLT F6</td>
<td>MLT F8</td>
<td></td>
</tr>
</tbody>
</table>

ADIR1,32
ADI R2,32
ADD F2 ADD F4
ADD F6 ADD F8
SGTR3
SD F2 SD F4
BEQZ
SD F6 SD F8

Total cycles = \(\frac{11}{4} = 2.75\) cycles
OVERVIEW OF THE NETBURST™ MICROARCHITECTURE

A fast processor requires balancing and tuning of many microarchitectural features that compete for processor die cost and for design and validation efforts. Figure 1 shows the basic Intel NetBurst microarchitecture of the Pentium 4 processor. As you can see, there are four main sections: the in-order front end, the out-of-order execution engine, the integer and floating-point execution units, and the memory subsystem.

In-Order Front End

The in-order front end is the part of the machine that fetches the instructions to be executed next in the program and prepares them to be used later in the machine pipeline. Its job is to supply a high-bandwidth stream of decoded instructions to the out-of-order execution core, which will do the actual completion of the instructions. The front end has highly accurate branch prediction logic that uses the past history of program execution to speculate where the program is going to execute next. The predicted instruction address, from this front-end branch prediction logic, is used to fetch instruction bytes from the Level 2 (L2) cache. These IA-32 instruction bytes are then decoded into basic operations called uops (micro-operations) that the execution core is able to execute.

The NetBurst microarchitecture has an advanced form of a Level 1 (L1) instruction cache called the Execution Trace Cache. Unlike conventional instruction caches, the Trace Cache sits between the instruction decode logic and the execution core as shown in Figure 1. In this location the Trace Cache is able to store the already decoded IA-32 instructions or uops. Storing already decoded instructions removes the IA-32 decoding from the main execution loop. Typically the instructions are decoded once and placed in the Trace Cache and then used repeatedly from there like a normal instruction cache on previous machines. The IA-32 instruction decoder is only used when the machine misses the Trace Cache and needs to go to the L2 cache to get and decode new IA-32 instruction bytes.

Out-of-Order Execution Logic

The out-of-order execution engine is where the instructions are prepared for execution. The out-of-order execution logic has several buffers that it uses to smooth and re-order the flow of instructions to optimize performance as they go down the pipeline and get scheduled for execution. Instructions are aggressively reordered to allow them to execute as quickly as their input operands are ready. This out-of-order execution allows instructions in the program following delayed instructions to proceed around them as long as they do not depend on those delayed instructions. Out-of-order execution allows the execution resources such as the ALUs and the cache to be kept as busy as possible executing independent instructions that are ready to execute.

The retirement logic is what reorders the instructions, executed in an out-of-order manner, back to the original program order. This retirement logic receives the completion status of the executed instructions from the execution units and processes the results so that the proper architectural state is committed (or retired) according to the program order. The Pentium 4 processor can retire up to three uops per clock cycle. This retirement logic ensures that exceptions occur only if the operation causing the exception is the oldest, non-retired operation in the machine. This logic also reports branch history information to the branch predictors at the front end of the machine so they can train with the latest known-good branch-history information.

Integer and Floating-Point Execution Units

The execution units are where the instructions are actually executed. This section includes the register files that store the integer and floating-point data operand values that the instructions need to execute. The execution units include several types of integer and floating-point execution units that compute the results and also the L1 data cache that is used for most load and store operations.

Memory Subsystem

Figure 1 also shows the memory subsystem. This includes the L2 cache and the system bus. The L2 cache stores both instructions and data that cannot fit in the Execution Trace Cache and the L1 data cache. The external system bus is connected to the backside of the second-level cache and is used to access main memory when the L2 cache has a cache miss, and to access the system I/O resources.
NETBURST MICROARCHITECTURE

Figure 4 shows a more detailed block diagram of the NetBurst microarchitecture of the Pentium 4 processor. The top-left portion of the diagram shows the front end of the machine. The middle of the diagram illustrates the out-of-order buffering logic, and the bottom of the diagram shows the integer and floating-point execution units and the L1 data cache. On the right of the diagram is the memory subsystem.

Front End

The front end of the Pentium 4 processor consists of several units as shown in the upper part of Figure 4. It has the Instruction TLB (ITLB), the front-end branch predictor (labeled here Front-End BTB), the IA-32 Instruction Decoder, the Trace Cache, and the Microcode ROM.