Advance Pipelining

- **Static Scheduling**
  - Have compiler to minimize the effect of structural, data, and control dependence
  - **advantages:** simple hardware
  - **Examples:**
    - Loop unrolling
    - Software Pipelining
    - Trace Scheduling

- **Dynamic Scheduling**
  - Have hardware to rearrange instruction execution to reduce the stalls
  - **advantages:**
    - handle dependence unknown at compile time
    - simplify compiler design
    - code compatible
  - **Examples:**
    - Scoreboarding
    - Tomasulo’s Algorithm
    - Dynamic Branch Prediction
## Instruction Level Parallelism

- **Taking advantages of ILP**
  - Superscalar - multiple issue per cycle
  - Superpipelining - deeper pipelines
  - VLIW - very long instruction word

- **Increase ILP by compiler**
  - Loop unrolling -
    Increase instructions between loop branch by replicating loop body multiple times
  - **Software Pipelining**
    Reorganize loop code such that each iteration contains code chosen from different iterations
  - **Trace Scheduling**
    Increase parallelism by selecting more code candidates

- **Usually hardware techniques require compiler support**
  - Superscalar needs instruction scheduling
  - VLIW needs trace scheduling

### Static Scheduling - Loop Unrolling

- **Basic Idea**
  - Loop: LD F0, 0(R1)
  - ADD F4, F0, F2
  - SUB R1, R1, #8
  - BNE R1, loop
  - SD 0(R1), F4

  Unrolled by 4

- **Features**
  - reduce loop overhead
    - branch
    - counter
  - increase code size
  - increase register requirement

- Loop: LD F0, 0(R1)
  - ADD F4, F0, F2
  - SD 0(R1), F4

- Unrolled by 4

- Loop: LD F6, 8(R1)
  - ADD F8, F6, F2
  - SD 8(R1), F8

- Loop: LD F10, 16(R1)
  - ADD F12, F10, F2
  - SD 16(R1), F12

- Loop: LD F14, 24(R1)
  - ADD F16, F14, F2
  - SD 24(R1), F16

- Loop: SUB R1, R1, #32
  - BNE R1, loop

© by Tien-Fu Chen@CCU
**Identify Dependencies by Compiler**

- **Goal:**
  - compiler concerned about dependencies in program, whether or not a HW hazard depends on a given pipeline
  - Data dependency (True)
  - Name dependency (anti-/output dependency)
  - Control dependency

- **(True) Data dependencies**
  - (RAW if a hazard for HW)
  - Instruction i produces a result used by instruction j, or
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
  - Easy to determine for registers (fixed names)
  - Hard for memory:
    - Does 100(R4) = 20(R6)?
    - From different loop iterations, does 20(R6) = 20(R6)?

**Name dependency**

- **Name dependence:**
  - two instructions use same name but don’t exchange data
  - Antidependence (WAR if a hazard for HW)
    - Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first
  - Output dependence (WAW if a hazard for HW)
    - Instruction i and instruction j write the same register or memory location; ordering between instructions must be preserved.
  - no value being transmitted between instructions

- **Register Renaming**
  - Statically by compiler
  - Dynamic by hardware
Control dependency

- control dependence

Example

if p1 {S1;};
if p2 {S2;}
- S1 is control dependent on p1 and
- S2 is control dependent on p2 but not on p1.

- Two constraints on control dependences:
  - An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.
  - An instruction that is not control dependent on a branch cannot be moved to after the branch so that its execution is controlled by the branch.

- Control dependencies relaxed to get parallelism; get same effect if
  - preserve exception behavior
  - preserve data flow

Loop-carried dependence

- Where are data dependencies?

  for (i=1; i<=100; i=i+1)
  {
    A[i+1] = A[i] + C[i]; /* S1 */
    B[i+1] = B[i] + A[i+1]; /* S2 */
  }
  1. S2 uses the value, A[i+1], computed by S1 in the same iteration.
  2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].

- Loop-carried dependence
  dependency exists between different iterations of the loop
  - Loop with loop-carried dependence cannot be executed in parallel

- GCD test:
  write A[a * i+b]
  read A[c * i+d]
  if loop-carried depend exists then
  (d - b) mod GCD(a,c) = 0
**Software Pipelining**

- **Software pipelining**
  
  Reorganize loops such that each iteration contains instruction sequences from different iterations in original code
  
  - need start-up & finish blocks

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

NEW:
for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

© by Tien-Fu Chen@CCU
Compared with loop unrolling

Before: Unrolled 3 times
1 L.D F0,0(R1)
2 ADD.D F4,F0,F2
3 S.D 0(R1),F4
4 L.D F6,-8(R1)
5 ADD.D F8,F6,F2
6 S.D -8(R1),F8
7 L.D F10,-16(R1)
8 ADD.D F12,F10,F2
9 S.D -16(R1),F12
10 DSUBUI R1,R1,#24
11 BNEZ R1,LOOP

After: Software Pipelined
1 S.D 0(R1),F4 ; Stores M[i]
2 ADD.D F4,F0,F2 ; Adds to M[i-1]
3 L.D F0,-16(R1); Loads M[i-2]
4 DSUBUI R1,R1,#8
5 BNEZ R1,LOOP

• Symbolic Loop Unrolling
  – Maximize result-use distance
  – Less code space than unrolling
  – Fill & drain pipe only once per loop
    vs. once per each unrolled iteration in loop unrolling

Trace Scheduling

- Trace
  a most likely sequence of instructions

- Two phases
  • trace selection
    - identify frequent codes
    - may use loop unrolling to generate long trace
    - should consider data dependence constraints and branch points
  • trace compaction
    - squeeze trace into a small of wide instructions
    - move instruction as early as possible
    - compact instructions as few as possible
    - add compensation code

- good or bad?
  • scheduling across basic blocks
  • code explosion
Summary of Static Scheduling

- Loop unrolling
  - multiple loop body for scheduling
  - reduce branch frequency
  - expand code size
  - must handle "residual" iterations
  - increase register pressure

- Software Pipelining
  - no dependences in loop body
  - does not reduce branch hazards
  - need start-up and finish blocks
  - increase register pressure

- Trace Scheduling
  - works for loops
  - increase most likely operations for VLIW
  - more complex than loop unrolling
  - code expansion

Advance Pipelining again

- Static Scheduling
  Have compiler to minimize the effect of structural, data, and control dependence

- Dynamic Scheduling
  HW rearrange the instruction execution to reduce stalls

- Key idea
  Allow instructions behind stall to proceed
  
  \[
  \text{DIVD F0,F2,F4} \\
  \text{ADDD F10,F0,F8} \\
  \text{SUBD F8,F8,F14}
  \]

  - Enables out-of-order execution => out-of-order completion
  - ID stage checked both for structural & data dependencies
  - Splitting ID into two stages
    - Issue - decode instructions, check for structural hazards
    - Read operands - wait until no data hazards, then read operands
Advantages of Dynamic Scheduling

- Handles cases when dependences unknown at compile time (e.g., because they may involve a memory reference)
- It simplifies the compiler
- Allows code that compiled for one pipeline to run efficiently on a different pipeline
- Hardware speculation, a technique with significant performance advantages, that builds on dynamic scheduling

Out-of-order execution

- Split pipelining stages:
  - Original DLX pipelining
  - New DLX pipelining

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID - decode, check for hazards, fetch operands</td>
</tr>
<tr>
<td></td>
<td>EX - execute instruction</td>
</tr>
<tr>
<td></td>
<td>WB - write back</td>
</tr>
</tbody>
</table>

- Issue - decode, check structural hazards
- read operands - wait until no data hazards
- execute
- write back

- Scoreboarding
  - multiple function units
  - each instruction goes through the scoreboard
  - centralized control scheme
    - control all instruction issue
    - detect all hazards
    - maintain hazard resolutions
  - implemented in CD 6600
Scoreboarding

- **Dealing with hazards**
  - **Issue**: check structural hazards; check for WAW hazards; stall issue until hazard cleared
  - **Read operand**: check for RAW hazards; wait until data ready
  - **Execution**: execute operations; notify scoreboard when completion
  - **Write back**: check for WAR; stall write until clear

- **Maintain three data structures**
  - **Instruction status**: indicate the four steps of instruction in
  - **Function unit status**: indicate states of each function units
  - **Register result status**: indicate which function unit will write a register

---

Three Parts of the Scoreboard

1. **Instruction status**—which of 4 steps the instruction is in

2. **Functional unit status**—Indicates the state of the functional unit (FU). 9 fields for each functional unit
   - Busy—Indicates whether the unit is busy or not
   - Op—Operation to perform in the unit (e.g., + or –)
   - Fi—Destination register
   - Fj, Fk—Source-register numbers
   - Qj, Qk—Functional units producing source registers Fj, Fk
   - Rj, Rk—Flags indicating when Fj, Fk are ready

3. **Register result status**—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions will write that register
**Detailed control and data maintaining**

<table>
<thead>
<tr>
<th>Instruction status</th>
<th>Wait until</th>
<th>Bookkeeping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue</td>
<td>Not busy (FU) and not result(D)</td>
<td>Busy(FU) ← yes; Op(FU) ← op; Fi(FU) ← 'D'; Fj(FU) ← 'S1'; Fk(FU) ← 'S2'; Qj ← Result('S1'); Qk ← Result('S2'); Rj ← not Qj; Rk ← not Qk; Result('D') ← FU;</td>
</tr>
<tr>
<td>Read operands</td>
<td>Rj and Rk</td>
<td>Rj ← No; Rk ← No</td>
</tr>
<tr>
<td>Execution complete</td>
<td>Functional unit done</td>
<td></td>
</tr>
<tr>
<td>Write result</td>
<td>∀f((Fj(f)≠Fi(FU)) or Rj(f)=No) &amp; (Fk(f) #Fi(FU) or Rk(f)=No))</td>
<td>∀f(if Qj(f)=FU then Rj(f)← Yes); ∀f(if Qk(f)=FU then Rk(f)← Yes); Result(Fi(FU)) ← 0; Busy(FU)← No</td>
</tr>
</tbody>
</table>

- Try to identify how scoreboard deals with
  - RAW
  - WAR
    - Queue both the operation and copies of its operands
    - Read registers only during Read Operands stage
  - WAW

© by Tien-Fu Chen@CCU
Scoreboarding (cont)

- **Benefits**
  - Improvement of 1.7 for Fortran programs
  - 2.5 for hand-coded assembly program
  - How about modern compiler??

- **Cost**
  - only as much as one function unit
  - main cost in buses (4 times)

- **Limitations**
  - Results are through register file, never data forwarded.
  - Data dependency must wait for access to register file
  - Stalls when WAW hazards occur
  - Limited to instructions in basic block (small window)

Tomasulo's Algorithm

- **Key features:**
  - Reservation stations at each functional unit
  - A Common Data Bus (CDB) to broadcast all results
  - Employ register renaming
  - Use "tag" to handle hazard control
  - load/store buffers

- **Only three steps (not include IF, MEM)**
  - Issue - get instruction
    - check available function unit
    - or check available load buffer
    - stalls on structural hazards
  - Execute - execute when operands available
    - if not, check CDB for operand
  - Write back - if CDB available, write result
    - if not, stalls on CDB

- **Data structures are attached to reservation stations, reg file, load buffer**
**Data structure of Tomasulo’s Algorithm**

**Reservation Station Components**

- **Op**—Operation to perform in the unit (e.g., + or –)

- **Vj, Vk**—Value of Source operands
  - Store buffers have V field, result to be stored

- **Qj, Qk**—Reservation stations producing source registers (value to be written)
  - Note: No ready flags as in Scoreboard; Qj,Qk=0 => ready
  - Store buffers only have Qi for RS producing result

- **Busy**—Indicates reservation station or FU is busy

- **Register result status**—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.
## Bookkeeping rules

<table>
<thead>
<tr>
<th>Instruction status</th>
<th>Wait until</th>
<th>Action or bookkeeping</th>
</tr>
</thead>
</table>
| Issue              | Station or buffer empty | if (Register['S1'].Qi ≠ 0)  
{RS[r].Qj ← Register['S1'].Qi}  
else {RS[r].Vj ← S1}; RS[r].Qj ← 0};  
if (Register[S2].Qi ≠ 0)  
{RS[r].Qk ← Register[S2].Qi};  
else {RS[r].Vk ← S2}; RS[r].Qk ← 0}  
RS[r].Busy ← yes;  
Register['D'].Qi ← r; |
| Execute            | (RS[r].Qj = 0) and (RS[r].Qk = 0) | None—operands are in Vj and Vk                                                                                                                                  |
| Write result       | Execution completed at r and CDB available | ∀x(if (Register[x].Qi = r)  
{Fx ← result;  
Register[x].Qi ← 0});  
∀x(if (RS[x].Qj = r)  
{RS[x].Vj ← result;  
RS[x].Qj ← 0});  
∀x(if (RS[x].Qk = r)  
{RS[x].Vk ← result;  
RS[x].Qk ← 0});  
∀x(if (Store[x].Qi = r)  
{Store[x].V ← result;  
Store[x].Qi ← 0});  
RS[r].Busy ← No |

### How to eliminating WAW and WAR?

© by Tien-Fu Chen@CCU

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

<table>
<thead>
<tr>
<th>Instruction status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue</td>
</tr>
<tr>
<td>Execute</td>
</tr>
<tr>
<td>Write result</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Vk</th>
<th>Qj</th>
<th>Qk</th>
<th>Reservation stations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add1</td>
<td>Yes</td>
<td>SUB</td>
<td>Mem[34+Regs[R2]]</td>
<td></td>
<td></td>
<td></td>
<td>Load2</td>
</tr>
<tr>
<td>Add2</td>
<td>Yes</td>
<td>ADD</td>
<td></td>
<td>Add1</td>
<td></td>
<td></td>
<td>Load2</td>
</tr>
<tr>
<td>Add3</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult1</td>
<td>Yes</td>
<td>MULT</td>
<td>Regs[F4]</td>
<td>Load2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult2</td>
<td>Yes</td>
<td>DIV</td>
<td>Mem[34+Regs[R2]]</td>
<td>Mult1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
</tr>
<tr>
<td>F0</td>
</tr>
<tr>
<td>F2</td>
</tr>
<tr>
<td>F4</td>
</tr>
<tr>
<td>F6</td>
</tr>
<tr>
<td>F8</td>
</tr>
<tr>
<td>F10</td>
</tr>
<tr>
<td>F12</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>F30</td>
</tr>
<tr>
<td>Qi</td>
</tr>
</tbody>
</table>
### Instruction status

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Execute</th>
<th>Write result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6,34(R2)</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>LD F2,45(R3)</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>MULTD F0,F2,F4</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>SUBD F8,F6,F2</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>DIVD F10,F0,F6</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F6,F8,F2</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>

### Reservation stations

<table>
<thead>
<tr>
<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>Add1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add3</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult1</td>
<td>Yes</td>
<td>MULT</td>
<td>Mem[45+Regs[R3]]</td>
<td>Regs[F4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult2</td>
<td>Yes</td>
<td>DIV</td>
<td>Mem[34+Regs[R2]]</td>
<td>Mult1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Register status

<table>
<thead>
<tr>
<th>Field</th>
<th>F0</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
<th>F12</th>
<th>...</th>
<th>F30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qi</td>
<td>Mult1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mult2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### With load/store buffers

#### Instruction status

<table>
<thead>
<tr>
<th>Instruction</th>
<th>From iteration</th>
<th>Issue</th>
<th>Execute</th>
<th>Write result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
<td>1</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTD F4,F0,F2</td>
<td>1</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>1</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F0,0(R1)</td>
<td>2</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>MULTD F4,F0,F2</td>
<td>2</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>2</td>
<td>√</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Reservation stations

<table>
<thead>
<tr>
<th>Name</th>
<th>Busy</th>
<th>Fm</th>
<th>Vj</th>
<th>Vk</th>
<th>Qj</th>
<th>Qk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add3</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult1</td>
<td>Yes</td>
<td>MULT</td>
<td>Mem[2]</td>
<td>Regs[F2]</td>
<td>Load1</td>
<td></td>
</tr>
</tbody>
</table>

#### Register status

<table>
<thead>
<tr>
<th>Field</th>
<th>F0</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
<th>F12</th>
<th>...</th>
<th>F30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q0</td>
<td>Load2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mult2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Load buffers

<table>
<thead>
<tr>
<th>Field</th>
<th>Load 1</th>
<th>Load 2</th>
<th>Load 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Regs[R1]</td>
<td>Regs[R1]-8</td>
<td></td>
</tr>
<tr>
<td>Busy</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

#### Store buffers

<table>
<thead>
<tr>
<th>Field</th>
<th>Store 1</th>
<th>Store 2</th>
<th>Store 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qi</td>
<td>Mult1</td>
<td>Mult2</td>
<td>No</td>
</tr>
<tr>
<td>Busy</td>
<td>Yes</td>
<td>Mult1</td>
<td>Mult2</td>
</tr>
<tr>
<td>Address</td>
<td>Regs[R1]</td>
<td>Regs[R1]-8</td>
<td></td>
</tr>
</tbody>
</table>
Tomasulo's Algorithm (cont)

- Differences from scoreboard:
  - Control & buffers distributed with Function Units,
  - HW renaming of registers to avoid WAW and WAR hazards
  - CDB to broadcast results rather than waiting on registers
  - Load/store buffers are treated as function unit

- Reservation Stations:
  - Handle distributed hazard detection and instruction control
  - Use 4-bit tag field to specify which RV station or load buffer will produce result

- Register renaming:
  - Tag assigned on instruction issue
  - Tag discarded after write back to register
  - WAW and WAR are eliminated
  - Even better if a branch-taken strategy on a loop
    multiple execution simultaneously
    loop is unrolled dynamically by hardware

Comparisons

**Scoreboaring**
- Global data structure centralized control
- Use register designator + simple, low cost
- Structural stalls on FU
- Solve:
  - RAW by register
  - WAR in write
  - WAW stalls on issue
- Limited:
  - register serialized input output
  - Stalls issue on block
    \[
    S1 \leftarrow S2 \cdot S3 \\
    S4 \leftarrow S5 \cdot S6
    \]

**Tomasulo's Alg**
- Distributed control
- Use tagged register + register renaming - tag allocation and deallocation - associative compare
- Structural stalls on RS
- Solve:
  - RAW by CDB
  - WAR copy opnd to RS
  - WAW by renaming
- Limited: CDB + broadcast result - one result per cycle
Dynamic Branch Prediction

- For a loop

Branch History Table
- Lower bits of PC address index table of 1-bit values
- Tells whether or not branch taken last time
- Problems:
  - the end of a loop and 1st time of next loop

2-bit scheme
where change prediction only if get misprediction twice.

Performance of Prediction Buffer

- Mispredict because either:
  - Wrong guess for that branch
  - Got history of wrong branch when index table

Prediction Accuracy
- 4096 entry table programs vary from 1% misprediction (nasa7, tomtat) to 18% (eqntott), with spice at 9% and gcc at 12%
- 4096 about as good as infinite table, but 4096 is a lot of HW

Performance of prediction buffer:
Given that 90% hit in buffer and 90% of correct guess

Accuracy = (%hit in buffer * %correct) + [(1 - %hit in buffer) * luck guess]
= (90% * 90%) + (50% * 10%)
= 86%
**Correlating Branch Predictor**

- **Idea:**
  - taken/not taken of recently executed branches is related to behavior of next branch (as well as the history of that branch behavior)

- **a (m,n) predictor**
  - use the behavior of the last m branches
  - choose from $2^m$ branch predictors
  - each is n-bit predictor for a single branch
  - cost:
    - $2^m \times n \times$ # of prediction entries selected by addr

**Branch Target Buffer**

- **Guess both branch cond and branch target**
  - IF: check BTB
  - ID: check cond
  - EX: refill BTB

- **Penalty**
  - | hit in buffer | prediction | actual branch | Penalty |
  - |---------|-----------|--------------|--------|
  - | Yes     | Taken    | Taken        | 0      |
  - | Yes     | Taken    | N Taken     | 2      |
  - | No      | Taken    |             | 2      |

- **Total penalty**
  - (60% taken branch)
  - $Penalty = \% \text{hit in buffer} * \% \text{incorrect} * 2 + (1 - \% \text{hit in buffer}) * \% \text{taken} * 2$
  - $= (90\% * 10\% * 2) + (10\% * 60\% * 2)$
  - $= 0.30 \text{ cycles}$
Variations of BTB

- Separating target buffer from a prediction buffer - PowerPC 620
  - target address prediction
  - branch direction

- Storing target instructions
  - allow longer time to access BTB, better for a larger BTB
  - allowed to perform branch folding

- Predicting indirect jumps
  - destination address varies at run-time
    e.g., indirect procedure calls, procedure returns,
  - return address stack

- Reducing misprediction penalty
  - fetching from both predicted and unpredicted direction ->requiring dual-ported instruction memory
  - caching instructions from multiple paths

Multiple-Issue Processors

- Superscalar
  - Issue varying # of instructions per clock
  - be either statically scheduled by compiler or dynamically scheduled using hardware
    e.g. IBM PowerPC, Sun SuperSparc, DEC Alpha, HP 7100

- Very Long Instruction Words (VLIW)
  - fixed number of instructions scheduled by the compiler
  - inherently statically scheduled by compiler
  - Joint HP/Intel agreement in 1998??

- Hardware support for more ILP
  - Conditional/Predicated Instructions
  - Speculative instructions with renaming
  - Speculative execution
Superscalar

- Superscalar machines
  - Issue multiple independent instructions per clock cycle
  - if dependence exist, only first instruction issued
  - hardware not dynamically concern about issue
  - need $n$ function units for degree of $n$
  - would help if compiler properly schedule codes

Limitation
- contention on FP registers
  - why: FP load/store vs FP operations
  - solution: dual-ported register file?
- Delayed load
  - why: original one delay slot, now 3 slots
- Delayed branch

Advantages:
- code compatibility
- could reach CPI < 1

Dynamic Scheduling in Superscalar

- Dependencies stop instruction issue
  - Code compiler for scalar version will run poorly on superscalar
  - May want code to vary depending on how superscalar

Simple approach:
- separate Tomasulo Control for separate reservation stations for Integer FU/Reg and for FP FU/Reg
- Issue 2X Clock Rate, so that issue remains in order
- Only FP loads might cause dependency between integer and FP issue:
  - Replace load reservation station with a load queue; operands must be read in the order they are fetched
  - Load checks addresses in Store Queue to avoid RAW violation
  - Store checks addresses in Load Queue to avoid WAR,WAW
**VLIW (Very Long Instruction Word)**

- **tradeoff instruction space for simple decoding**
  - long instruction word has room for many operations
  - all the operations the compiler puts in the long instruction word can execute in parallel
    - E.g., 2 int ops, 2 FP ops, 2 mems, 1 branch
  - 16 to 24 bits per field => 7*16 = 112 bits to 7*24 = 168 bits wide

- **Features**:  
  - Central controller issues a single long instruction  
  - Each instruction initiates many independent operations  
  - Each operation executed in fixed cycles  
  - Operations can be pipelined

- **Compiler Aid**
  - loop unrolling  
  - scheduling code across basic blocks  
    - trace scheduling

---

**Limitations on multiple-issue**

- **Limited ILP**
  - 1 branch in 5 instructions => how to keep a 5-way VLIW busy?  
  - Latencies of units => many operations must be scheduled  
  - Need Pipeline Depth x No. Functional Units of independent operations to keep machines busy

- **Limits on VLIW**
  - Limited amount of parallelism available in instruction sequences  
  - require a large-number of memory and register bandwidth for different functional units at the same time  
  - code size explosion  
    - loop unrolling + no op code  
  - VLIW lock step => 1 hazard & all instructions stall  
  - binary compatibility is practical weakness
Comparisons

Superscalar

adv. :
- better code density
- code compatible

difference :
- issue decision at: Run Time

disadv :
- require more instruction fetching & decoding
- more delayed slot to be filled
- different functional units
- muti-ported reg. file
- limited parallelism

VLIW

adv. :
- fixed instruction format
- more parallelism provided
- by trace scheduling at: Compile Time

disadv :
- static scheduling - compiler
- no dynamic decision
- code explosion
- multi-ported register file
- limited parallelism
- difficult to use data cache because of sharing

Hardware Support for more ILP

- Conditional (predicate) instructions
  - Eliminate branches by turning branches into conditionally executed instructions
    if (x) then A=B op C else NOP
  - If false, then neither store result or cause exception
  - Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can nullify any following instructions

- Limitation of Conditional Branches
  - Still takes a clock even if annulled
  - Stall if condition evaluated late
  - Limited use when control flow involves more than a simple sequence; Complex conditions reduce effectiveness, condition becomes known late in pipeline
  - May increase CPI for conditional instructions or cause a slower overall clock rate
Speculation

- Speculation:
  - Allow an instruction to issue that is dependent on branch predicted to be taken without any consequences (including exceptions) if branch is not actually taken (without undo)
  - Methods support ambitious speculation

- HW/SW cooperation
  - HW and OS handle exceptions and return an undefined value for any exception
  - may not be acceptable

- Poison bits
  - a poison bit added to each register and instruction
  - a fault occurs on using a value from poison reg

- Speculation instruction with renaming
  - boosting instruction by flagging as speculation and providing renaming and buffering

Hardware-based Speculation

- Basic idea
  - Allow instruction executed out of order but force them to commit in order; prevent irrevocable action until commits
  - separate speculative bypassing of results from real bypassing of results

- Introducing Reorder Buffer
  - buffer results of uncommitted instructions
  - pass results among speculated instructions
  - 3 fields: instruction, type, destination, and value
  - key differences from Tomasulo’s Algorithm:
    - Tomasulo: read result from register once result is written
    - reorder buffer: supply operands between completion and commit time
  - replace load/store buffer
  - Use reorder buffer number instead of reservation station buffer
Speculative Tomasulo’s Algorithm

- **Four steps of Tomasulo’s algorithm**
  - **Issue**
    If reservation station or reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination.
  - **Execution**
    When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute
  - **Write result**
    Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.
  - **Commit**
    When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer.

Speculative DLX using Tomasulo’s Algorithm
Register renaming, virtual registers versus Reorder Buffers

- Alternative to Reorder Buffer is a larger virtual set of registers and register renaming.

- Virtual registers hold both architecturally visible registers + temporary values:
  - replace functions of reorder buffer and reservation station.

- Renaming process maps names of architectural registers to registers in virtual register set:
  - Changing subset of virtual registers contains architecturally visible registers.

- Simplifies instruction commit: mark register as no longer speculative, free register with old value.

- Adds 40-80 extra registers: Alpha, Pentium,...
  - Size limits no. instructions in execution (used until commit).

Limitation of ILP

- Hardware model of perfect processors:
  - Register renaming: infinite virtual registers and all WAW & WAR hazards are avoided.
  - Branch prediction: perfect; no mispredictions.
  - Jump prediction: jumps perfectly predicted => machine with perfect speculation & an unbounded buffer of instructions available.
  - Memory-address alias analysis: addresses are known & a store can be moved before a load provided addresses not equal.
  - 1 cycle latency for all instructions.

- measurement: IPC:
  - instruction issues per cycle.

- Observation:
  - FP programs are more sensitive to limit window size.
## Workstation Microprocessors 3/2001

<table>
<thead>
<tr>
<th>Processor</th>
<th>Alpha 21264B</th>
<th>AMD Athlon</th>
<th>HP PA-8600</th>
<th>IBM Power2-II</th>
<th>Intel Pentium III</th>
<th>Intel Pentium 4</th>
<th>MIPS R12000</th>
<th>Sun Ultra-II</th>
<th>Sun Ultra-III</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clock Rate</strong></td>
<td>833MHz</td>
<td>1.2GHz</td>
<td>552MHz</td>
<td>450MHz</td>
<td>1.0GHz</td>
<td>1.5GHz</td>
<td>400MHz</td>
<td>480MHz</td>
<td>900MHz</td>
</tr>
<tr>
<td><strong>Cache (L1/D/L2)</strong></td>
<td>64K/64K</td>
<td>64K/64K/256K</td>
<td>512K/1M</td>
<td>32K/64K</td>
<td>16K/16K/256K</td>
<td>12K/2K/256K</td>
<td>32K/32K</td>
<td>16K/16K</td>
<td>32K/64K</td>
</tr>
<tr>
<td><strong>Issue Rate</strong></td>
<td>4 issue</td>
<td>3 x 86 instr</td>
<td>4 issue</td>
<td>4 issue</td>
<td>3 x 86 instr</td>
<td>3 x ROOps</td>
<td>4 issue</td>
<td>4 issue</td>
<td>4 issue</td>
</tr>
<tr>
<td><strong>Pipeline Stages</strong></td>
<td>7/9 stages</td>
<td>9/11 stages</td>
<td>7/9 stages</td>
<td>7/8 stages</td>
<td>12/14 stages</td>
<td>22/24 stages</td>
<td>6 stages</td>
<td>6/9 stages</td>
<td>14/15 stages</td>
</tr>
<tr>
<td><strong>Out of Order</strong></td>
<td>80 instr</td>
<td>72ROPs</td>
<td>56 instr</td>
<td>32 instr</td>
<td>40 ROOps</td>
<td>126 ROOps</td>
<td>48 instr</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td><strong>Rename regs</strong></td>
<td>48/41</td>
<td>36/36</td>
<td>56 total</td>
<td>16 int/24 fp</td>
<td>40 total</td>
<td>128 total</td>
<td>32/32</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td><strong>BHT Entries</strong></td>
<td>4K x 9-bit</td>
<td>4K x 2-bit</td>
<td>2K x 2-bit</td>
<td>2K x 2-bit</td>
<td>&gt;= 512</td>
<td>4K x 2-bit</td>
<td>2K x 2-bit</td>
<td>512 x 2-bit</td>
<td>16K x 2-bit</td>
</tr>
<tr>
<td><strong>TLB Entries</strong></td>
<td>128/128</td>
<td>280/288</td>
<td>120 unified</td>
<td>128/128</td>
<td>32/48D</td>
<td>128/56D</td>
<td>64 unified</td>
<td>64/64D</td>
<td>128/512D</td>
</tr>
<tr>
<td><strong>Memory B/W</strong></td>
<td>2.66GB/s</td>
<td>2.1GB/s</td>
<td>1.5GB/s</td>
<td>1.6GB/s</td>
<td>1.06GB/s</td>
<td>3.2GB/s</td>
<td>539 MB/s</td>
<td>1.5GB/s</td>
<td>4.8GB/s</td>
</tr>
<tr>
<td><strong>Package</strong></td>
<td>CGA-58/s</td>
<td>CGA-46/s</td>
<td>LGA-544</td>
<td>SCC-1088</td>
<td>PGA-370</td>
<td>PGA-423</td>
<td>CGA-527</td>
<td>CLGA-778</td>
<td>1368 FC-LGA</td>
</tr>
<tr>
<td><strong>IC Process</strong></td>
<td>0.18μm</td>
<td>0.18μm</td>
<td>0.25μm</td>
<td>0.22μm</td>
<td>0.18μm</td>
<td>0.18μm</td>
<td>0.25μm</td>
<td>0.29μm</td>
<td>0.18μm</td>
</tr>
<tr>
<td><strong>Die Size</strong></td>
<td>115mm²</td>
<td>117mm²</td>
<td>477mm²</td>
<td>163mm²</td>
<td>106mm²</td>
<td>217mm²</td>
<td>204mm²</td>
<td>126mm²</td>
<td>210mm²</td>
</tr>
<tr>
<td><strong>Transistors</strong></td>
<td>15.4 million</td>
<td>37 million</td>
<td>130 million</td>
<td>23 million</td>
<td>24 million</td>
<td>42 million</td>
<td>7.2 million</td>
<td>3.8 million</td>
<td>29 million</td>
</tr>
<tr>
<td><strong>Est mfg cost</strong></td>
<td>$160</td>
<td>$62</td>
<td>$330</td>
<td>$110</td>
<td>$39</td>
<td>$110</td>
<td>$125</td>
<td>$70</td>
<td>$145</td>
</tr>
<tr>
<td><strong>PowerMax</strong></td>
<td>75W*</td>
<td>76W</td>
<td>60W*</td>
<td>36W*</td>
<td>30W</td>
<td>55W</td>
<td>25W*</td>
<td>20W*</td>
<td>65W</td>
</tr>
<tr>
<td><strong>Availability</strong></td>
<td>1Q01</td>
<td>4Q00</td>
<td>3Q00</td>
<td>4Q00</td>
<td>2Q00</td>
<td>4Q00</td>
<td>2Q00</td>
<td>3Q00</td>
<td>4Q00</td>
</tr>
</tbody>
</table>

Max issue: 4 instructions (many CPUs)
Max rename registers: 128 (Pentium 4)
Max BHT: 4K x 9 (Alpha 21264B), 16Kx2 (Ultra III)
Max Window Size (OOO): 126 instructions (Pent. 4)
Max Pipeline: 22/24 stages (Pentium 4)

Source: Microprocessor Report, www.MPRonline.com

---

## SPEC 2000 Performance 3/2001

<table>
<thead>
<tr>
<th>Processor</th>
<th>Alpha 21264B</th>
<th>AMD Athlon</th>
<th>HP PA-8600</th>
<th>IBM Power3</th>
<th>Intel P4</th>
<th>MIPS R12000</th>
<th>Sun Enterprise 450</th>
<th>Sun Blade 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System or Motherboard</strong></td>
<td>Alpha E540 Model 2</td>
<td>AMD 72ZM</td>
<td>HP5000 j6000</td>
<td>RS6000 4PP-170</td>
<td>Dell Prec. 420</td>
<td>Intel D850GB</td>
<td>Sun 2200</td>
<td>Sun Blade 1000</td>
</tr>
<tr>
<td><strong>Clock Rate</strong></td>
<td>833MHz</td>
<td>1.2GHz</td>
<td>552MHz</td>
<td>450MHz</td>
<td>1.5GHz</td>
<td>400MHz</td>
<td>480MHz</td>
<td>500MHz</td>
</tr>
<tr>
<td><strong>External Cache</strong></td>
<td>8MB</td>
<td>None</td>
<td>None</td>
<td>8MB</td>
<td>None</td>
<td>8MB</td>
<td>8MB</td>
<td>8MB</td>
</tr>
<tr>
<td><strong>164_gzip</strong></td>
<td>392</td>
<td>n/a</td>
<td>376</td>
<td>230</td>
<td>564</td>
<td>527</td>
<td>226</td>
<td>165</td>
</tr>
<tr>
<td><strong>175_vpr</strong></td>
<td>452</td>
<td>n/a</td>
<td>421</td>
<td>285</td>
<td>354</td>
<td>298</td>
<td>384</td>
<td>212</td>
</tr>
<tr>
<td><strong>176_gcc</strong></td>
<td>617</td>
<td>n/a</td>
<td>577</td>
<td>350</td>
<td>401</td>
<td>588</td>
<td>313</td>
<td>232</td>
</tr>
<tr>
<td><strong>181_mcf</strong></td>
<td>441</td>
<td>n/a</td>
<td>384</td>
<td>498</td>
<td>276</td>
<td>473</td>
<td>563</td>
<td>356</td>
</tr>
<tr>
<td><strong>186_crafty</strong></td>
<td>694</td>
<td>n/a</td>
<td>472</td>
<td>304</td>
<td>523</td>
<td>497</td>
<td>334</td>
<td>175</td>
</tr>
<tr>
<td><strong>197_parser</strong></td>
<td>360</td>
<td>n/a</td>
<td>361</td>
<td>171</td>
<td>362</td>
<td>472</td>
<td>283</td>
<td>211</td>
</tr>
<tr>
<td><strong>252_eon</strong></td>
<td>645</td>
<td>n/a</td>
<td>395</td>
<td>280</td>
<td>615</td>
<td>650</td>
<td>360</td>
<td>209</td>
</tr>
<tr>
<td><strong>253_perlbmk</strong></td>
<td>526</td>
<td>n/a</td>
<td>406</td>
<td>215</td>
<td>614</td>
<td>703</td>
<td>246</td>
<td>247</td>
</tr>
<tr>
<td><strong>254_gap</strong></td>
<td>365</td>
<td>n/a</td>
<td>229</td>
<td>256</td>
<td>463</td>
<td>708</td>
<td>204</td>
<td>171</td>
</tr>
<tr>
<td><strong>255_vortex</strong></td>
<td>673</td>
<td>n/a</td>
<td>764</td>
<td>312</td>
<td>717</td>
<td>725</td>
<td>294</td>
<td>304</td>
</tr>
<tr>
<td><strong>256_bzip2</strong></td>
<td>560</td>
<td>n/a</td>
<td>349</td>
<td>258</td>
<td>396</td>
<td>420</td>
<td>334</td>
<td>237</td>
</tr>
<tr>
<td><strong>300_twolf</strong></td>
<td>658</td>
<td>n/a</td>
<td>479</td>
<td>414</td>
<td>394</td>
<td>403</td>
<td>451</td>
<td>243</td>
</tr>
<tr>
<td><strong>SPECint_base2000</strong></td>
<td>516</td>
<td>n/a</td>
<td>417</td>
<td>285</td>
<td>494</td>
<td>524</td>
<td>320</td>
<td>225</td>
</tr>
<tr>
<td><strong>168_wupsid</strong></td>
<td>529</td>
<td>360</td>
<td>340</td>
<td>360</td>
<td>417</td>
<td>795</td>
<td>280</td>
<td>284</td>
</tr>
<tr>
<td><strong>171_swim</strong></td>
<td>1,156</td>
<td>506</td>
<td>761</td>
<td>279</td>
<td>493</td>
<td>1,244</td>
<td>300</td>
<td>285</td>
</tr>
<tr>
<td><strong>172_mgrid</strong></td>
<td>580</td>
<td>272</td>
<td>462</td>
<td>319</td>
<td>274</td>
<td>558</td>
<td>231</td>
<td>225</td>
</tr>
<tr>
<td><strong>173_appiu</strong></td>
<td>424</td>
<td>298</td>
<td>563</td>
<td>327</td>
<td>280</td>
<td>641</td>
<td>237</td>
<td>150</td>
</tr>
<tr>
<td><strong>177_mesa</strong></td>
<td>713</td>
<td>302</td>
<td>300</td>
<td>330</td>
<td>541</td>
<td>553</td>
<td>289</td>
<td>273</td>
</tr>
<tr>
<td><strong>178_galgel</strong></td>
<td>558</td>
<td>468</td>
<td>569</td>
<td>429</td>
<td>335</td>
<td>537</td>
<td>989</td>
<td>735</td>
</tr>
<tr>
<td><strong>179_art</strong></td>
<td>1,540</td>
<td>213</td>
<td>419</td>
<td>669</td>
<td>410</td>
<td>514</td>
<td>995</td>
<td>920</td>
</tr>
<tr>
<td><strong>183_equake</strong></td>
<td>231</td>
<td>236</td>
<td>347</td>
<td>560</td>
<td>289</td>
<td>739</td>
<td>222</td>
<td>149</td>
</tr>
<tr>
<td><strong>187_facorec</strong></td>
<td>822</td>
<td>411</td>
<td>258</td>
<td>257</td>
<td>307</td>
<td>451</td>
<td>411</td>
<td>459</td>
</tr>
<tr>
<td><strong>188_anmp</strong></td>
<td>488</td>
<td>221</td>
<td>376</td>
<td>326</td>
<td>294</td>
<td>366</td>
<td>373</td>
<td>313</td>
</tr>
<tr>
<td><strong>189_lucas</strong></td>
<td>731</td>
<td>237</td>
<td>370</td>
<td>284</td>
<td>349</td>
<td>764</td>
<td>259</td>
<td>205</td>
</tr>
<tr>
<td><strong>191_fma3d</strong></td>
<td>528</td>
<td>365</td>
<td>302</td>
<td>540</td>
<td>297</td>
<td>427</td>
<td>192</td>
<td>207</td>
</tr>
<tr>
<td><strong>200_sixtrack</strong></td>
<td>340</td>
<td>256</td>
<td>286</td>
<td>234</td>
<td>170</td>
<td>257</td>
<td>199</td>
<td>159</td>
</tr>
<tr>
<td><strong>301_aspi</strong></td>
<td>553</td>
<td>278</td>
<td>523</td>
<td>349</td>
<td>373</td>
<td>427</td>
<td>252</td>
<td>189</td>
</tr>
<tr>
<td><strong>SPECfp_base2000</strong></td>
<td>590</td>
<td>304</td>
<td>400</td>
<td>356</td>
<td>325</td>
<td>549</td>
<td>319</td>
<td>274</td>
</tr>
</tbody>
</table>

Source: Microprocessor Report, www.MPRonline.com
**Dynamic Scheduling in Intel Pentium II, III**

Q: How pipeline 1 to 17 byte 80x86 instructions?

- Pentium does not pipeline 80x86 instructions
- Its decode unit translates the Intel instructions into 72-bit micro-operations (~ MIPS)
- Sends micro-operations to reorder buffer & reservation stations
- Many instructions translate to 1 to 4 micro-operations
- Complex 80x86 instructions are executed by a conventional microprogram (8K x 72 bits) that issues long sequences of micro-operations
- 14 clocks in total pipeline (~ 3 state machines)

---

**Dynamic Scheduling in Intel Pentium**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>80x86 micro-ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. instructions issued/clock</td>
<td>3</td>
</tr>
<tr>
<td>Max. instr. complete exec./clock</td>
<td>5</td>
</tr>
<tr>
<td>Max. instr. commited/clock</td>
<td>3</td>
</tr>
<tr>
<td>Window (Instrs in reorder buffer)</td>
<td>40</td>
</tr>
<tr>
<td>Number of reservations stations</td>
<td>20</td>
</tr>
<tr>
<td>Number of rename registers</td>
<td>40</td>
</tr>
<tr>
<td>No. integer functional units (FUs)</td>
<td>2</td>
</tr>
<tr>
<td>No. floating point FUs</td>
<td>1</td>
</tr>
<tr>
<td>No. SIMD Fl. Pt. FUs</td>
<td>1</td>
</tr>
<tr>
<td>No. memory Fus</td>
<td>1 load + 1 store</td>
</tr>
</tbody>
</table>
Pipelining in Pentium

- 14 clocks in total (~3 state machines)
- 8 stages are used for in-order instruction fetch, decode, and issue
  - Takes 1 clock cycle to determine length of 80x86 instructions + 2 more to create the micro-operations (uops)
- 3 stages are used for out-of-order execution in one of 5 separate functional units
- 3 stages are used for instruction commit

Pentium block diagram
Pentium III Die Photo

- EBL/BBL - Bus logic, Front, Back
- MOB - Memory Order Buffer
- Packed FPU - MMX Fl. Pt. (SSE)
- IEU - Integer Execution Unit
- FAU - Fl. Pt. Arithmetic Unit
- MIU - Memory Interface Unit
- DCU - Data Cache Unit
- PMH - Page Miss Handler
- DTLB - Data TLB
- BAC - Branch Address Calculator
- RAT - Register Alias Table
- SIMD - Packed Fl. Pt.
- RS - Reservation Station
- BTB - Branch Target Buffer
- IFU - Instruction Fetch Unit (+I$)
- ID - Instruction Decode
- ROB - Reorder Buffer
- MS - Micro-instruction Sequencer