Lecture 18: Instruction Level Parallelism
-- Dynamic Scheduling, Multiple Issue, and Speculation

CSE 564 Computer Architecture Summer 2017

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan
### Topics for Instruction Level Parallelism

- **ILP Introduction, Compiler Techniques and Branch Prediction**
  - 3.1, 3.2, 3.3

- **Dynamic Scheduling (OOO)**
  - 3.4, 3.5 and C.5, C.6 and C.7 (FP pipeline and scoreboard)

- **Hardware Speculation and Static Superscalar/VLIW**
  - 3.6, 3.7

- **Dynamic Scheduling, Multiple Issue and Speculation**
  - 3.8, 3.9, 3.13

- **ILP Limitations and SMT**
  - 3.10, 3.11, 3.12
Acknowledgement and Copyright

- Slides adapted from
  - UC Berkeley course “Computer Science 252: Graduate Computer Architecture” of David E. Culler Copyright(C) 2005 UCB
  - UC Berkeley course Computer Science 252, Graduate Computer Architecture Spring 2012 of John Kubiatowicz Copyright(C) 2012 UCB
  - Computer Science 152: Computer Architecture and Engineering, Spring 2016 by Dr. George Michelogiannakis from UC Berkeley

- https://passlab.github.io/CSE564/copyrightack.html
REVIEW
Not Every Stage Takes only one Cycle

- **FP EXE Stage**
  - **Multi-cycle Add/Mul**
  - **Nonpiplined for DIV**

- **MEM Stage**

---

*Figure C.41* The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating...
Issues of Multi-Cycle in Some Stages

- The divide unit is not fully pipelined
  - structural hazards can occur
    » need to be detected and stall incurred.
- The instructions have varying running times
  - the number of register writes required in a cycle can be > 1
- Instructions no longer reach WB in order
  - Write after write (WAW) hazards are possible
    » Note that write after read (WAR) hazards are not possible, since the register reads always occur in ID.
- Instructions can complete in a different order than they were issued (out-of-order complete)
  - causing problems with exceptions
- Longer latency of operations
  - stalls for RAW hazards will be more frequent.
Hardware Solution for Addressing Data Hazards

- **Dynamic Scheduling**
  - Out-of-order execution and completion

- **Data Hazard via Register Renaming**
  - Dynamic RAW hazard detection and scheduling in data-flow fashion
  - Register renaming for WRW and WRA hazard (name conflict)

- **Implementations**
  - **Scoreboard (CDC 6600 1963)**
    » Centralized register renaming
  - **Tomasulo’s Approach (IBM 360/91, 1966)**
    » Distributed control and renaming via reservation station, load/store buffer and common data bus (data+source)
Register Renaming Summary

- **Purpose of Renaming**: removing “Anti-dependencies”
  - Get rid of WAR and WAW hazards, since these are not “real” dependencies

- **Implicit Renaming**: i.e. Tomasulo
  - Registers changed into values or response tags
  - We call this “implicit” because space in register file may or may not be used by results!

- **Explicit Renaming**: more physical registers than needed by ISA.
  - Rename table: tracks current association between architectural registers and physical registers
  - Uses a translation table to perform compiler-like transformation on the fly

- **With Explicit Renaming**:
  - All registers concentrated in single register file
  - Can utilize bypass network that looks more like 5-stage pipeline
  - Introduces a register-allocation problem
    - Need to handle branch misprediction and precise exceptions differently, but ultimately makes things simpler
HARDWARE SPECULATION: ADDRESSING CONTROL HAZARDS
Control Hazard from Branches: Three Stage Stall if Taken

10: BEQ R1,R3,36
14: AND R2,R3,R5
18: OR R6,R1,R7
22: ADD R8,R1,R9
36: XOR R10,R1,R11

What do you do with the 3 instructions in between? How do you do it?
### Control Hazards

- **Break the instruction flow**
- **Unconditional Jump**
- **Conditional Jump**
- **Function call and return**
- **Exceptions**

Multiple instructions following branch in program order can complete before branch resolves.
Branch Prediction + Speculation
(Misprediction Recovery)

- **Branch Prediction:**
  - Modern branch predictors have high accuracy: (>95%) and can reduce branch penalties significantly
  - Required hardware support
    » Branch history tables (Taken or Not)
    » Branch target buffers, etc. (Target address)

- **In-order commit for out-of-order execution:**
  - Instructions fetched and decoded into instruction reorder buffer in-order
  - Execution is out-of-order (⇒ out-of-order completion)
  - Commit (write-back to architectural state, i.e., regfile & memory) is in-order
Branch Prediction/Speculation

Branch Prediction

Fetch

Decode & Rename

Reorder Buffer

PC

Branch Resolution

Update predictors

Commit

Reg. File

Branch Unit

ALU

MEM

Store Buffer

D$

Execute

kill

kill

kill

kill
Hardware Speculation in Tomasulo Algorithm

- Reservation Station and Load Buffer
  - Register renaming
  - For dynamic scheduling and out-of-order execution

- Reorder Buffer
  - Register renaming
  - For in-order commit

- Common Data Bus
  - Data forwarding

- Also handle memory data hazard
Four Steps of *Speculative* Tomasulo

1. **Issue**—get instruction from FP Op Queue
   
   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called “dispatch”)

2. **Execution**—operate on operands (EX)
   
   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called “issue”)

3. **Write result**—finish execution (WB)
   
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.

4. **Commit**—update register with reorder result
   
   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called “graduation”)

---

**Notes:**
- FP Op Queue: Floating Point Operations Queue
- FUs: Functional Units
- RAW: Ready to Access Register
- CDB: Common Data Bus
- Mispredicted branch: Branch prediction error

---

**Key Terms:**
- *Speculative*: Predicting the outcome of a computation to avoid waiting for results
- Tomasulo: Algorithm for efficient instruction processing
- Reservation station: Holds instructions temporarily
- Reorder buffer: Stores results for later use
- Dispatch: Initial stage of instruction execution
- Issue: The process of sending instructions to the execution units
- Execution: Processing of instructions
- Write: Writing results back to registers
- Commit: Finalizing the execution of instructions
Instruction In-order Commit

- Also called completion or graduation
- In-order commit
  - In-order issue
  - Out-of-order execution
  - Out-of-order completion
- Three cases when an instr reaches the head of ROB
  - Normal commit: when an instruction reaches the head of the ROB and its result is present in the buffer
    » The processor updates the register with the result and removes the instruction from the ROB.
  - Committing a store:
    » is similar except that memory is updated rather than a result register.
  - A branch with incorrect prediction
    » indicates that the speculation was wrong.
    » The ROB is flushed and execution is restarted at the correct successor of the branch.
In-order Commit with Branch

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

<table>
<thead>
<tr>
<th>Entry</th>
<th>Busy</th>
<th>Instruction</th>
<th>State</th>
<th>Destination</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No</td>
<td>L.D F0,0(R1)</td>
<td>Commit</td>
<td>F0</td>
<td>Mem[0 + Regs[R1]]</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>MUL.D F4,F0,F2</td>
<td>Commit</td>
<td>F4</td>
<td>#1 × Regs[F2]</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>S.D F4,0(R1)</td>
<td>Write result</td>
<td>0 + Regs[R1]</td>
<td>#2</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>DADDIU R1,R1,#-8</td>
<td>Write result</td>
<td>R1</td>
<td>Regs[R1] – 8</td>
</tr>
<tr>
<td>5</td>
<td>Yes</td>
<td>BNE R1,R2,Loop</td>
<td>Write result</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Yes</td>
<td>L.D F0,0(R1)</td>
<td>Write result</td>
<td>F0</td>
<td>Mem[#4]</td>
</tr>
<tr>
<td>7</td>
<td>Yes</td>
<td>MUL.D F4,F0,F2</td>
<td>Write result</td>
<td>F4</td>
<td>#6 × Regs[F2]</td>
</tr>
<tr>
<td>8</td>
<td>Yes</td>
<td>S.D F4,0(R1)</td>
<td>Write result</td>
<td>0 + #4</td>
<td>#7</td>
</tr>
<tr>
<td>9</td>
<td>Yes</td>
<td>DADDIU R1,R1,#-8</td>
<td>Write result</td>
<td>R1</td>
<td>#4 – 8</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>BNE R1,R2,Loop</td>
<td>Write result</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FP register status

<table>
<thead>
<tr>
<th>Field</th>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reorder #</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>Busy</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>...</td>
</tr>
</tbody>
</table>

FLUSHED

IF Misprediction
Dynamic Scheduling and Speculation

- **ILP Maximized (a restricted data-flow)**
  - In-order issue
  - Out-of-order execution
  - Out-of-order completion
  - In-order commit

- **Data Hazards**
  - Input operands-driven dynamic scheduling for RAW hazard
  - Register renaming for handling WAR and WAW hazards

- **Control Hazards (Branching, Precision Exception)**
  - Branch prediction and in-order commit (speculation)
  - Branch prediction without speculation
    » Cannot do out-of-order execution/complete for branch

- **Implementation: Tomasulo**
  - Reservation stations and Reorder buffer
  - Other solutions as well (scoreboard, history table)
MULTIPLE ISSUE VIA VLIW/STATIC SUPERSCALAR
Multiple Issue

- “Flynn bottleneck”
  - single issue performance limit is \( CPI = IPC = 1 \)
  - hazards + overhead \( \Rightarrow CPI \geq 1 \) (\( IPC \leq 1 \))
  - diminishing returns from superpipelining \([\text{Hrishikesh paper!}]\)

- Solution: issue multiple instructions per cycle

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst0</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst2</td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>inst3</td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
</tr>
</tbody>
</table>

- 1st superscalar: IBM America \( \rightarrow \) RS/6000 \( \rightarrow \) POWER1
Multiple Issue

- Issue multiple instructions in one cycle
- Three major types (VLIW and superscalar)
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - Dynamically scheduled superscalar processors
- Superscalar
  - Variable # of instr per cycle
  - In-order execution for static superscalar
  - Out-of-order execution for dynamic superscalar
- VLIW
  - Issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallel-ism among instructions explicitly indicated by the instruction.
  - Inherently statically scheduled by the compiler
  - Intel/HP IA-64 architecture, named EPIC—explicitly parallel instruction computer
  » Appendix H,
VLIW and Static Superscalar

- Very similar in terms of the requirements for compiler and hardware support
- We will discuss VLIW/Static superscalar

- Very Long Instruction Word (VLIW)
  - packages the multiple operations into one very long instruction
VLIW: Very Large Instruction Word

- Each “instruction” has explicit coding for multiple operations
  - In IA-64, grouping called a “packet”
  - In Transmeta, grouping called a “molecule” (with “atoms” as ops)

- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 1 integer operation/branch, 2 FP ops, 2 Memory refs
    » 16 to 24 bits per field => 5*16 or 80 bits to 5*24 or 120 bits wide
  - Need compiling technique that schedules across several branches
Recall: Unrolled Loop that Minimizes Stalls for Scalar

for \((i=999; i>=0; i=i-1)\)

\[ x[i] = x[i] + s; \]

1 Loop: 
1. L.D F0,0(R1)  
2. L.D F6,-8(R1)  
3. L.D F10,-16(R1)  
4. L.D F14,-24(R1)  
5. ADD.D F4,F0,F2  
6. ADD.D F8,F6,F2  
7. ADD.D F12,F10,F2  
8. ADD.D F16,F14,F2  
9. S.D 0(R1),F4  
10. S.D -8(R1),F8  
11. S.D -16(R1),F12  
12. DSUBUI R1,R1,#32  
13. BNEZ R1,LOOP  
14. S.D 8(R1),F16  

L.D to ADD.D: 1 Cycle  
ADD.D to S.D: 2 Cycles

14 clock cycles, or 3.5 per iteration

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another 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>

; 8–32 = −24
Loop Unrolling in VLIW

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
Average: 2.5 ops per clock, 50% efficiency

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

**Figure 3.16** VLIW instructions that cycles assuming no branch delay;Sorations in 9 clock cycles, or 2.5 operation, is about 60%. To achieve thi this loop. The VLIW code sequence and MIPS processor can use as few as two l

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another 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>
Loop Unrolling in VLIW

- **Unroll 8 times**
  - **Enough registers**

  8 results in 9 clocks, or 1.125 clocks per iteration

  *Average: 2.89 (26/9) ops per clock, 58% efficiency (26/45)*

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

**Figure 3.16** VLIW instructions that cycles assuming no branch delay; narrations in 9 clock cycles, or 2.5 operations. To achieve this loop, the VLIW code sequence above MIPS processor can use as few as two!

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another 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>
### Loop Unrolling in VLIW

- **Unroll 10 times**
  - **Enough registers**

10 results in 10 clocks, or 1 clock per iteration

**Average:** 3.2 ops per clock (32/10), 64% efficiency (32/50)

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

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another 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>
Problems with 1st Generation VLIW

- Increase in code size
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding

- Operated in lock-step; no hazard detection HW
  - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might prediction function units, but caches hard to predict

- Binary code compatibility
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code

- Itanium/EPIC/VLIW is not a breakthrough in ILP
  - If anything, it is as complex or more so than a dynamic processor
Very Important Terms

- **Dynamic Scheduling** → Out-of-order Execution
- **Speculation** → In-order Commit
- **Superscalar** → Multiple Issue

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Goals</th>
<th>Implementation</th>
<th>Addressing</th>
<th>Approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Scheduling</td>
<td>Out-of-order execution</td>
<td>Reservation Stations, Load/Store Buffer and CDB</td>
<td>Data hazards (RAW, WAW, WAR)</td>
<td>Register renaming</td>
</tr>
<tr>
<td>Speculation</td>
<td>In-order commit</td>
<td>Reorder Buffer</td>
<td>Control hazards</td>
<td>Prediction and misprediction recovery</td>
</tr>
<tr>
<td>Superscalar/ VLIW</td>
<td>Multiple issue</td>
<td>Software and Hardware</td>
<td>To Increase CPI</td>
<td>By compiler or hardware</td>
</tr>
</tbody>
</table>
DYNAMIC SCHEDULING, MULTIPLE ISSUE (DYNAMIC SUPERSCALAR), AND SPECULATION
Dynamic Scheduling, Multiple Issue, and Speculation

- Microarchitecture quite similar to those in modern microprocessors
  - Real

\[
\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}
\]

- Consider two issue per clock
  - Example: CPU with floating point ALUs: Issue 1 FP + 1 Integer instruction per cycle.
    - Save at least 1 cycle than the pipeline
  - Challenges
    - Find the right instructions
    - Dependency between instructions
what is involved in
- fetching two instructions per cycle?
- decoding two instructions per cycle?
- executing two ALU operations per cycle?
- accessing the data cache twice per cycle?
- writing back two results per cycle?

what about 4 or 8 instructions per cycle?
Implementation using Temasulo’s Approach

- Similar to Tomasulo with Speculation

- Multiple issue → one issue per clock cycle per functional unit
  - 4-wide
Options and Challenges of Multiple Issue

- How to issue two instructions and keep in-order instruction issue for Tomasulo?
  - Assume 1 integer + 1 floating point
  - 1 Tomasulo control for integer, 1 for floating point

1. Issue two instrs pipelined in one cycle (half and half for each instr), so that issue remains in order → superpipelining
   - Hard to extend to 4 or more

2. Issue 2 instrs per cycle in parallel → true superscalar
   - Between FP and Integer operations: 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
     » Called “decoupled architecture”

3. Mix of both
   - Superpipelining and superscalar
Multiple Issue Challenges

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards

- If more instructions issue at same time, greater difficulty of decode and issue:
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue
  - Multiported rename logic: must be able to rename same register multiple times in one cycle!
  - Rename logic one of key complexities in the way of multiple issue!
Multiple Issue

- Bundle multiple instrs in one issue unit
  - \( N \)-wide

1. Assign a reservation station and a reorder buffer for every instruction that *might* be issued in the next issue bundle
   - \( N \) entries in ROB
   - Ensure enough RS available for the bundle
   - If not enough RS/ROB, break the bundle

2. Analyze dependency in the issue bundle

3. Inter-dependency between instrs in a bundle
   - Update the reservation station table entries using the assigned ROB entries to link the dependency
     » Register renaming happened

- In-order commit to make sure instrs commit in order
- Other techniques
  - Speculative multiple issue in Intel i7
Example

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

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

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

BNE has RAW dependence on DADDIU
**Without Speculation**

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Instructions</th>
<th>Issues at clock cycle number</th>
<th>Executes at clock cycle number</th>
<th>access at clock cycle number</th>
<th>Write CDB at clock cycle number</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD R2,0(R1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>First issue</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R2,R2,#1</td>
<td>1</td>
<td>5</td>
<td></td>
<td>6</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>1</td>
<td>SD R2,0(R1)</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R1,R1,#8</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td>Execute directly</td>
</tr>
<tr>
<td>1</td>
<td>BNE R2,R3,LOOP</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
<td>No Speculation</td>
</tr>
<tr>
<td>2</td>
<td>LD R2,0(R1)</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R2,R2,#1</td>
<td>4</td>
<td>11</td>
<td></td>
<td>12</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>2</td>
<td>SD R2,0(R1)</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>14</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R1,R1,#8</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td></td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>2</td>
<td>BNE R2,R3,LOOP</td>
<td>6</td>
<td>13</td>
<td></td>
<td></td>
<td>No Speculation</td>
</tr>
<tr>
<td>3</td>
<td>LD R2,0(R1)</td>
<td>7</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R2,R2,#1</td>
<td>7</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>3</td>
<td>SD R2,0(R1)</td>
<td>8</td>
<td>15</td>
<td>19</td>
<td>19</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R1,R1,#8</td>
<td>8</td>
<td>14</td>
<td></td>
<td>15</td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>3</td>
<td>BNE R2,R3,LOOP</td>
<td>9</td>
<td>19</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
</tbody>
</table>

Figure 3.19: The time of issue, execution, and writing result for a dual-issue version of our pipeline without speculation. Note that the LD following the BNE cannot start execution earlier because it must wait until the branch outcome is determined. This type of program, with data-dependent branches that cannot be resolved earlier, shows...
With Speculation

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Instructions</th>
<th>Issues at clock number</th>
<th>Execute at clock number</th>
<th>at clock number</th>
<th>clock</th>
<th>at clock number</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD R2,0(R1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>First issue</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R2,R2,#1</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>1</td>
<td>SD R2,0(R1)</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>Commit in order</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R1,R1,#8</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>Commit in order</td>
</tr>
<tr>
<td>1</td>
<td>BNE R2,R3,LOOP</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>LD R2,0(R1)</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>No execute delay</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R2,R2,#1</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>2</td>
<td>SD R2,0(R1)</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R1,R1,#8</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>Commit in order</td>
</tr>
<tr>
<td>2</td>
<td>BNE R2,R3,LOOP</td>
<td>6</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>LD R2,0(R1)</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>Earliest possible</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R2,R2,#1</td>
<td>7</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>3</td>
<td>SD R2,0(R1)</td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R1,R1,#8</td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>Executes earlier</td>
</tr>
<tr>
<td>3</td>
<td>BNE R2,R3,LOOP</td>
<td>9</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
</tbody>
</table>

Figure 3.20 The time of issue, execution, and writing result for a dual-issue version of our pipeline with speculation. Note that the LD following the BNE can start execution early because it is speculative.
ADVANCED TECHNIQUES FOR INSTRUCTION DELIVERY AND SPECULATION

1. Advanced Branch Prediction
2. Explicit Register Renaming
3. Others that are important but not covered: Load/store speculation, value predication, correlate branch prediction, tournament predictor, trace cache
4. Put all together on ARM Cortex-A8 and Intel Core i7
BRANCH PREDICTION BEFORE DECODING
Branch History Table for Dynamic Branch Prediction

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

  - Red: stop, not taken;
  - Blue: go, taken;
  - Adds *hysteresis* to decision making process.

![Branch History Table Diagram](image-url)
Typical Branch History Table

Fetch PC

I-Cache

Instruction

Opcode

offset

Branch?

Target PC

Taken/¬Taken?

2^k-entry BHT, n bits/entry

4K-entry BHT, 2 bits/entry, ~80-90% correct predictions
BP bits are stored with the predicted target address. (BHT)

IF stage: If (BP=taken) then nPC=target else nPC=PC+4
later: check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb
Branch Target Buffer (BTB)

- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *predicted taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded
Consulting BTB Before Decoding

- The match for PC=1028 fails and 1028+4 is fetched
  \(\Rightarrow\) eliminates false predictions after ALU instructions

- BTB contains entries only for control transfer instructions
  \(\Rightarrow\) more room to store branch targets
Branch Misprediction in Pipeline

- Can have multiple unresolved branches in ROB
- Can resolve branches out-of-order by killing all the instructions in ROB that follow a mispredicted branch
Branch With a Target Buffer

- **Steps**

```
<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Enter branch instruction address and next PC into branch-target buffer</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Branch correctly predicted; continue execution with no stalls</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Send PC to memory and branch-target buffer</td>
</tr>
<tr>
<td>Yes</td>
<td></td>
<td>Entry found in branch-target buffer?</td>
</tr>
<tr>
<td>No</td>
<td></td>
<td>Is instruction a taken branch?</td>
</tr>
<tr>
<td>Yes</td>
<td></td>
<td>Send out predicted PC</td>
</tr>
<tr>
<td>No</td>
<td></td>
<td>Normal instruction execution</td>
</tr>
</tbody>
</table>
```


Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

fa() { fb(); nexta: }
fb() { fc(); nextb: }
fcc() { fd(); nextc: }

Push return address when function call executed

Pop return address when subroutine return decoded

k entries (typically k=8-16)
Special Case Return Addresses

- Register Indirect branch hard to predict address
  - SPEC89 85% such branches for procedure return
  - Since stack discipline for procedures, save return address in small buffer that acts like a stack: 8 to 16 entries has small miss rate
Performance: Return Address Predictor

- **Cache most recent return addresses:**
  - Call ⇒ Push a return address on stack
  - Return ⇒ Pop an address off stack & predict as new PC
Independent “Fetch” unit

- Instruction fetch decoupled from execution
  - Instruction Buffer in-between
- Often issue logic (+ rename) included with Fetch

Stream of Instructions To Execute

Instruction Fetch with Branch Prediction

Out-Of-Order Execution Unit

Correctness Feedback On Branch Results
EXPLICIT REGISTER RENAMING
Explicit Register Renaming

- Tomasulo provides *Implicit Register Renaming*
  - User registers renamed to reservation station tags

- **Explicit Register Renaming:**
  - Use *physical* register file that is larger than number of registers specified by ISA

- **Keep a translation table:**
  - ISA register => physical register mapping
  - When register is written, replace table entry with new register from freelist.
  - Physical register becomes free when not being used by any instructions in progress.

- **Pipeline can be exactly like “standard” DLX pipeline**
  - IF, ID, EX, etc....

- **Advantages:**
  - Removes all WAR and WAW hazards
  - Like Tomasulo, good for allowing full out-of-order completion
  - Allows data to be fetched from a single register file
  - Makes speculative execution/precise interrupts easier:
    » All that needs to be “undone” for precise break point is to undo the table mappings
Explicit Renaming Support Includes:

- Rapid access to a table of translations
- A physical register file that has more registers than specified by the ISA
- Ability to figure out which physical registers are free.
  - No free registers $\Rightarrow$ stall on issue
- Thus, register renaming doesn’t require reservation stations. However:
  - Many modern architectures use explicit register renaming + Tomasulo-like reservation stations to control execution.
Explicit Register Renaming

- Make use of a *physical* register file that is larger than number of registers specified by ISA

- Keep a translation table:
  - ISA register => physical register mapping
  - When register is written, replace table entry with new register from freelist.
  - Physical register becomes free when not being used by any instructions in progress.
Explicit register renaming:
R10000 Freelist Management

- Physical register file larger than ISA register file
- On issue, each instruction that modifies a register is allocated new physical register from freelist
- Used on: R10000, Alpha 21264, HP PA8000
Explicit register renaming:
R10000 Freelist Management


Current Map Table

Freelist

| P34 | P36 | P38 | P40 |  ... | P60 | P62 |

Done?

Newest

Oldest

- Note that physical register P0 is “dead” (or not “live”) past the point of this load.
  - *When we go to commit the load, we free up*
Explicit register renaming:
R10000 Freelist Management

Current Map Table

Freelist

P32 P2 P4 P6 P8 P34 P12 P14 P16 P18 P20 P22 P24 P26 P28 P30

Done?

Newest

Oldest

P36 P38 P40 P42 \ldots P60 P62

F10 P10 ADDD P34, P4, P32 N
F0 P0 LD P32, 10(R2) N
Explicit register renaming:
R10000 Freelist Management

Current Map Table

<table>
<thead>
<tr>
<th>P32</th>
<th>P36</th>
<th>P4</th>
<th>F6</th>
<th>F8</th>
<th>P34</th>
<th>P12</th>
<th>P14</th>
<th>P16</th>
<th>P18</th>
<th>P20</th>
<th>P22</th>
<th>P24</th>
<th>P26</th>
<th>P28</th>
<th>P30</th>
</tr>
</thead>
</table>

Freelist

<table>
<thead>
<tr>
<th>P38</th>
<th>P40</th>
<th>P44</th>
<th>P48</th>
<th>...</th>
<th>P60</th>
<th>P62</th>
</tr>
</thead>
</table>

Done?

<table>
<thead>
<tr>
<th>--</th>
<th>BNE P36,&lt;...&gt;</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>DIV P36,P34,P6  N</td>
<td></td>
</tr>
<tr>
<td>F10 P10</td>
<td>ADDO P34,P4,P32  N</td>
<td></td>
</tr>
<tr>
<td>F0</td>
<td>P0</td>
<td>LD P32,10 (R2)  N</td>
</tr>
</tbody>
</table>

Checkpoint at BNE instruction
Explicit register renaming:
R10000 Freelist Management

Current Map Table

Freeclist

Done?

Newest

Oldest

Checkpoint at BNE instruction
Explicit register renaming:
R10000 Freelist Management

Current Map Table

Done?

Newest

Oldest

Current Map Table

Done?

Freelist

Error fixed by restoring map table and *merging* freelist

Checkpoint at BNE instruction

Error fixed by restoring map table and *merging* freelist
Advantages of Explicit Renaming

- **Decouples renaming from scheduling:**
  - Pipeline can be exactly like “standard” DLX pipeline (perhaps with multiple operations issued per cycle)
  - Or, pipeline could be tomasulo-like or a scoreboard, etc.
  - Standard forwarding or bypassing could be used

- Allows data to be fetched from single register file
  - No need to bypass values from reorder buffer
  - This can be important for balancing pipeline

- Many processors use a variant of this technique:
  - R10000, Alpha 21264, HP PA8000

- Another way to get precise interrupt points:
  - All that needs to be “undone” for precise break point is to undo the table mappings
  - Provides an interesting mix between reorder buffer and future file
    » Results are written immediately back to register file
    » Registers *names* are “freed” in program order (by ROB)
Superscalar Register Renaming

- During decode, instructions allocated new physical destination register
- Source operands renamed to physical register with newest value
- Execution unit only sees physical register numbers

![Diagram of Superscalar Register Renaming](image)

**Update Mapping**

**Rename Table**

**Register Free List**

**Read Addresses**

**Read Data**

**Write Ports**

**Inst 1**

**Inst 2**
Superscalar Register Renaming (Try #2)

Must check for RAW hazards between instructions issuing in same cycle. Can be done in parallel with rename lookup.

MIPS R10K renames 4 serially-RAW-dependent insts/cycle
Reality and References

- Modern processors use the advanced technologies we talked about in this class and others not covered
  - Principles are the same mostly

- Historically and more depth
  - Lots of ideas have been evaluated and developed
  - Appendix L.5 for history and references
  - VLIW/EPIC and software pipelining: Appendix H

- More and Latest Info (Conference)
  - MICRO: Annual IEEE/ACM International Symposium on Microarchitecture
    » https://www.microarch.org
  - IEEE Symposium on High Performance Computer Architecture (HPCA)
    » http://hpca2017.org/
  - International Symposium on Computer Architecture (ISCA)
  - ACM International Conference on Architectural Support for Programming Languages and Operating Systems
    » http://www.ece.cmu.edu/calcm/asplos2016
  - SIGARCH – The ACM Special Interest Group on Computer Architecture
    » https://www.sigarch.org/
Put It All Together Examples: ARM Cortex-A8 and Intel Core i7

- ARM Cortex-A8 core, the basis for the Apple A9 processor in the iPad, iPhones 3GS and 4
  - Dual-issue, statically scheduled superscalar with dynamic issue detection \( \rightarrow 0.5 \) CPI ideally
  - The basic pipeline structure of the 13-stage pipeline.

- F0 F1 F2 D0 D1 D2 D3 D4 E0 E1 E2 E3 E4 E5

- Instruction fetch:
  - AGU
  - RAM + TLB
  - BTB
  - QHB
  - RS

- Instruction decode:
  - 12-entry fetch queue

- Branch mispredict penalty = 13 cycles

- Instruction execute and load/store:
  - Architectural register file
  - ALU/MUL pipe 0
  - ALU pipe 1
  - LS pipe 0 or 1

Figure 3.36 The basic structure of the A8 pipeline is 13 stages. Three cycles are used for instruction fetch and four for instruction decode, in addition to a five-cycle integer pipeline. This yields a 13-cycle branch misprediction penalty. The instruction fetch unit tries to keep the 12-entry instruction queue filled.
ARM Cortex-A8 core, the basis for the Apple A9 processor in the iPad, iPhones 3GS and 4

- A dynamic branch predictor with a 512-entry two-way set associative branch target buffer and a 4K-entry global history buffer, indexed by the branch history and the current PC.
- An eight-entry return stack is kept to track return addresses.
  » Misprediction results in a 13-cycle penalty of the pipeline flushed.
Cortex-A8 Decode

- 5-stage decoding

![Diagram of 5-stage decoding]

**Figure 3.37** The five-stage instruction decode of the A8. In the first stage, a PC produced by the fetch unit (either from the branch target buffer or the PC incrementer) is used to retrieve an 8-byte block from the cache. Up to two instructions are decoded and placed into the decode queue; if neither instruction is a branch, the PC is incremented for the next fetch. Once in the decode queue, the scoreboard logic decides when the instructions can issue. In the issue, the register operands are read; recall that in a simple scoreboard, the operands always come from the registers. The register operands and opcode are sent to the instruction execution portion of the pipeline.
Cortex-A8 EXE Stage

- 5-stage execution (E5 is actually WB)

### Diagram

- **Architectural register file**
- **Shft** → **ALU + flags** → **Sat** → **BP update** → **WB**
- **MUL 1** → **MUL 2** → **MUL 3** → **ACC** → **WB**
- **ALU pipe 0**
- **ALU pipe 1**
- **Load/store pipe 0 or 1**

**Instruction execute**
- Integer register write back

**FLOW**

- INST 0
- INST 1
Intel Core i7

- Aggressive out-of-order speculative
- 14 stages pipeline,
- Branch mispredictions costing 17 cycles.
- 48 load and 32 store buffers.
- Six independent functional units
  - 6-wide superscalar
Instruction fetch – Fetch 16 bytes from the I cache

- A multilevel branch target buffer to achieve a balance between speed and prediction accuracy.
- A return address stack to speed up function return.
- Mispredictions cause a penalty of about 15 cycles.
Predecode –16 bytes instr in the predecode I buffer

- *Macro-op fusion:* Fuse instr combinations such as compare followed by a branch into a single operation.
- *Instr break down:* breaks the 16 bytes into individual x86 instructions.
  - nontrivial since the length of an x86 instruction can be from 1 to 17 bytes and the predecoder must look through a number of bytes before it knows the instruction length.
- Individual x86 instructions (including some fused instructions) are placed into the 18-entry instruction queue.
Core i7 Pipeline: Micro-op decode

- **Micro-op decode** – Translate Individual x86 instructions into micro-ops.
  - Micro-ops are simple MIPS-like instructions that can be executed directly by the pipeline (RISC style)
    » introduced in the Pentium Pro in 1997 and has been used since.
  - Three simple micro-op decoders handle x86 instructions that translate directly into one micro-op.
  - One complex micro-op decoder produce the micro-op sequence of complex x86 instr;
    » produce up to four micro-ops every cycle
  - The micro-ops are placed according to the order of the x86 instructions in the 28-entry micro-op buffer.
Core i7 Pipeline: loop stream detection and microfusion

- **loop stream detection and microfusion by the micro-op buffer preforms**
  - If there is a sequence of instructions (less than 28 instrs or 256 bytes in length) that comprises a loop, the loop stream detector will find the loop and directly issue the micro-ops from the buffer
    » eliminating the need for the instruction fetch and instruction decode stages to be activated.
  - Microfusion combines instr pairs such as load/ALU operation and ALU operation/store and issues them to a single reservation station, thus increasing the usage of the buffer.
    » Study comparing the microfusion and macrofusion by Bird et al. [2007] discovered that microfusion had little impact on performance, while macrofusion appears to have a modest positive impact on integer performance and little impact on FP.
Core i7 Pipeline: Issue

- Basic instruction issue
  - Looking up the register location in the register tables
  - renaming the registers
  - allocating a reorder buffer entry
  - fetching any results from the registers or reorder buffer before send reservation stations.

- 36-entry centralized reservation station shared by six functional units
  
  Up to six micro-ops may be dispatched to the functional units every clock cycle.
Micro-ops are executed by the individual function units

- results are sent back to any waiting reservation station as well as to the register retirement unit, where they will update the register state. The entry corresponding to the instruction in the reorder buffer is marked as complete.

Retirement

- When one or more instructions at the head of the reorder buffer have been marked as complete, the pending writes in the register retirement unit are executed, and the instructions are removed from the reorder buffer.
Core i7 Performance

- The integer CPI values range from 0.44 to 2.66 with a standard deviation of 0.77.
- The FP CPU is from 0.62 to 1.38 with a standard deviation of 0.25.
- Cache behavior is major contribution to the stall CPI.