이 강좌는 C & S Technology 사의 지원으로 제작되었으며 copyright가 없으므로 비영리적인 목적으로 한하여 누구든지 복사, 배포가 가능합니다. 연구실 홈페이지에는 고성능 마이크로프로세서에 관련된 많은 강좌가 있으며 누구나 무료로 다운로드 받을 수 있습니다.

고성능 마이크로프로세서
수퍼스칼라 구조의 개요
(High Performance Microprocessor Superscalar Architecture Overview)
2002. 8. 20
연세대학교 전기전자공학과
이용석 교수
Homepage: http://mpu.yonsei.ac.kr
E-mail: yonglee@yonsei.ac.kr
전화: 02-392-7194

Reference

History
Non - pipelined (8086, 286) 1970s
Pipelined (386, 486) 1980s
Superscalar (Pentium) VLIW 1990s
Parallel Architecture SMT : Simultaneous Multi - Threading 2000s
**Superscalar \( \mu P \) (Ref [2])**

- Executes multiple instructions simultaneously (ILP)
- Compiler minimizes data dependency
- Hardware instruction scheduling
- Most superscalar \( \mu Ps \) can execute 2-6 instructions per cycle
- Compatibility

**ILP (Instruction Level Parallelism)**
- Superscalar

**TLP (Thread Level Parallelism)**
- Multiprocessor
- SMT (Ref [2])

**Five Stage Pipeline (Ref. [1])**

F: Inst fetch (inst cache access) & inc PC
D: Inst decode & reg read
E: ALU op or addr calculation
M: Data cache access
W: Reg writeback

**Scalar \( \mu P \)**

1 inst per clock (IPC)

**2 - Way Superscalar \( \mu P \)**

2 IPC maximum

**4 - Way Superscalar \( \mu P \)**

4 IPC maximum
**Superscalar Instruction Issue Example (Alpha 21264)**

- 4 way superscalar
  - 4 integer
  - 2 floating point
  - 2 load or store

**Data Dependencies (Ref. [2], p96)**

- RAR (Read After Read) - no dependency
- RAW (Read After Write) - true dependency
- WAR (Write After Read) - false dependency
- WAW (Write After Write) - false dependency

**RAW (True dependency)**

\[
\begin{align*}
R_5 & \leftarrow R_2 + R_3 \\
R_5 & \leftarrow R_4 + R_4
\end{align*}
\]

- Cannot be issued simultaneously in superscalar

**Data Bypassing (Forwarding)**

\[
\begin{align*}
R_5 & \leftarrow R_2 + R_3 \\
R_5 & \leftarrow R_4 + R_4
\end{align*}
\]

**WAR (False dependency)**

\[
\begin{align*}
R_3 & \leftarrow R_2 + R_2 \\
R_5 & \leftarrow R_4 + R_5
\end{align*}
\]

- Cannot happen in the 5-stage pipeline
- Register renaming

**WAW (False dependency)**

\[
\begin{align*}
R_3 & \leftarrow R_2 + R_3 \\
R_5 & \leftarrow R_4 + R_5
\end{align*}
\]

- Register renaming
**Register Renaming**

* Compiler
  \[ R_1 \leftarrow R_2 + R_3 \]
  \[ R_6 \leftarrow R_4 + R_5 \]
* Hardware
  \[ R_1 \leftarrow R_2 + R_3 \]
  \[ R_1' \leftarrow R_4 + R_5 \]

**Superscalar Compiler**

* Compatibility
* Optimized for superscalar

**Superscalar Branch Strategy**

* 20% branch frequency

  \[
  \begin{align*}
  \text{Branch} & \quad 4 \sim 6 \text{ issues} \\
  \text{Branch} & \\
  \text{Branch} & \\
\end{align*}
  \]

**Superscalar Branch Strategy**

* Delayed branch,
  predict - not - taken \( X \)
* BTB (Branch Target Buffer)
* Loop unrolling
* Software pipelining

**Delayed Branch in Scalar**

- Compiler has to fill the delay slot

**Delayed Branch in Superscalar**

- Address
**Predict – Not – Taken in Scalar**

- Not taken branch
  - I + 1
  - I + 2

- Taken branch
  - I + 1
  - Branch target

**Predict – Not – Taken in Superscalar**

- I - 1
- I: not taken branch
  - I + 1
  - I + 2
  - I + 3
  - I + 4

---

**BTB (Branch Target Buffer)**

- Fully associative
- Pentium’s BTB: 4 way & 256 entries

**Branch History Bits (2 bits)**

- Predict Taken 0 0
  - Taken
  - Not Taken

- Predict Taken 0 1
  - Taken
  - Not Taken

- Predict Not Taken 1 1
  - Taken
  - Not Taken

- Predict Not Taken 1 0
  - Taken
  - Not Taken

**Loop:**

- i = 10
- if i ≠ 0 branch to Loop

- (1) i = i - 1
- (2) i
- (3) -

---

5
**Prediction by BTB (Ref [1])**

Result of compare

- **I-1**: compare
- **I**: branch
- **I+1**: branch target by BTB
- **I+2**: branch target + 1
- **I+3**: correct target

**Address input to BTB**

**From BTB**

**Address bus**

**Verification**

- **Flushed if mispredicted!**

---

**4 - Way Superscalar**

Result of compare

- **Compare**
- **Branch**
- **Branch target by BTB**
- **Correct branch target**

**Loop Unrolling**

Loop: 

\[ i = 10 \]

\[ i = i - 1 \]

if \( i \neq 0 \) branch to Loop

---
**Speed – up in Loop Unrolling**

- No branch instructions
- Large basic block size
  → more efficient scheduling

---

**Software Pipelining (Ref [2])**

```plaintext
for i=1 to 5 do {  
  a(i) = 2.0 * b(i)  
}
```

```plaintext
load R1, b(i)  
fmul R2, 2.0, R1  
dec R3  
store a(i), R2
```

- load, store - 1 cycle
- fmul - 3 cycles
- dec - 1 cycle

---

### Cycle | Iteration
--- | ---
C₀ | 1 | 2 | 3 | 4 | 5
C₁ | 2 | fmul | 3 | load
C₂ | 4 | dec | 5 | fmul | 6 | load
C₃ | | dec | 7 | fmul | 8 | load
C₄ | 9 | store | 10 | dec | fmul | load
C₅ | store | | | | | 
C₆ | | store | | | | 
C₇ | | | | | | 
C₈ | | | | | | 

---

### Compiler Hardware

```plaintext
  1 load  C₀  
  2 fmul  C₁  
  3 load  
  4 dec  
  5 fmul  
  6 load  
  7 dec  
  8 fmul  
  9 load  
  10 store  
  :  
```

---

**Further Issues**

- Reservation Station
- Scoreboard
- Out of order issue