Pipelined Architecture
with solutions
to data & control hazards
Pipeline Processing Hazards

- Structural Hazard
  - hardware duplication

- Data Hazard
  - Pipeline Stall
  - Software (machine code) optimization
  - Forwarding

- Control Hazard
  - Pipeline Flush (Instruction Invalidation)
  - Delayed Branching
  - Early Branch Detection
  - Branch History Table
Pipeline Stall

Some stages must be repeated – other invalidated

Reading the register being modified:

the same register can be referred to in ID (read) and in WB (write) stage – the writing can be done before (half clock cycle) reading

A) ADD R1, R2, R3
B) SUB R4, R3, R5
C) MUL R4, R3, R1
D) ...
E) ...
F) ...

<table>
<thead>
<tr>
<th>time</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td>B</td>
<td></td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>C</td>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>F</td>
<td>E</td>
<td>D</td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
</tr>
</tbody>
</table>
Hardware Pipeline Stall

- Deactivation (→0) of control signals for stages: Ex, Mem and WB
- Postponed writing to PC and IF/ID
Software "Pipeline Stall"

Software correction of data flow with NOP (No Operation)

not truly optimization, but might be occasionally necessary when hardware mechanisms are insufficient

```
next: LW   R1,0(R3)
MUL   R1,R1,R1
SW    R1,0(R3)
SUBI  R3,#4,R3
BNE   R0,R3,next

... all the data hazards "solved" with NOPs
```

```
next: LW   R1,0(R3)
NOP
MUL   R1,R1,R1
NOP
NOP
SW    R1,0(R3)
SUBI  R3,#4,R3
NOP
NOP
BNE   R0,R3,next

... all the data hazards "solved" with NOPs
```
Software Optimization for Architecture

- Static: optimization at compilation time (optimising compiler)
  - e.g. gcc -O2 -march=xxx

- Dynamic: at run-time: executing instructions in optimal order detected by hardware
  - dynamic scheduling
  - rename registers
  - out of order execution
  - speculative execution

Beyond the scope of this lecture
GCC Settings

gcc -o test test.c -O3 -march=athlon

<table>
<thead>
<tr>
<th>Target CPU Types</th>
<th>-march= Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i386 DX/SX/CX/EX/SL</td>
<td>i386</td>
</tr>
<tr>
<td>i486 DX/SX/DX2/SL/SX2/DX4</td>
<td>i486</td>
</tr>
<tr>
<td>487</td>
<td>i486</td>
</tr>
<tr>
<td>Pentium</td>
<td>pentium</td>
</tr>
<tr>
<td>Pentium MMX</td>
<td>pentium-mmx</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>pentiumpro</td>
</tr>
<tr>
<td>Pentium II</td>
<td>pentium2</td>
</tr>
<tr>
<td>Celeron</td>
<td>pentium2</td>
</tr>
<tr>
<td>Pentium III</td>
<td>pentium3</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>pentium4</td>
</tr>
<tr>
<td>Via C3</td>
<td>c3</td>
</tr>
<tr>
<td>Winchip 2</td>
<td>winchip2</td>
</tr>
<tr>
<td>Winchip C6-2</td>
<td>winchip-c6</td>
</tr>
<tr>
<td>AMD K5</td>
<td>i586</td>
</tr>
<tr>
<td>AMD K6</td>
<td>k6</td>
</tr>
<tr>
<td>AMD K6 II</td>
<td>k6-2</td>
</tr>
<tr>
<td>AMD K6 III</td>
<td>k6-3</td>
</tr>
<tr>
<td>AMD Athlon</td>
<td>athlon</td>
</tr>
<tr>
<td>AMD Athlon 4</td>
<td>athlon</td>
</tr>
<tr>
<td>AMD Athlon XP/MP</td>
<td>athlon</td>
</tr>
<tr>
<td>AMD Duron</td>
<td>athlon</td>
</tr>
<tr>
<td>AMD Tbird</td>
<td>athlon-tbird</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimization</th>
<th>-O1</th>
<th>-O2</th>
<th>-Os</th>
<th>-O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>defer-pop</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>thread-jumps</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>branch-probabilities</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>cprocr-registers</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>guess-branch-probability</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>omit-frame-pointer</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>align-loops</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>align-jumps</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>align-labels</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>align-functions</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>optimize-sibling-calls</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>cse-follow-jumps</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>cse-skip-blocks</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>gccoe</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>expensive-optimizations</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>strength-reduce</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>rerun-cse-after-loop</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>rerun-loop-opt</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>caller-saves</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>force-men</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>peephole2</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>regmove</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>strict-aliasing</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>delete-null-pointer-checks</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>reorder-blocks</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>schedule-insns</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>schedule-insns2</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>inline-functions</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>rename-functions</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Included in Level
Static Optimization Example

For pipelined architecture with *Single Delay Slot*

```
next: LW   R1,0(R3)
MUL   R1,R1,R1
SW    R1,0(R3)
SUBI  R3,#4,R3
BNE   R0,R3,next
...
```

```
next: LW   R1,0(R3)
SUBI  R3,#4,R3
MUL   R1,R1,R1
BNE   R0,R3,next
SW    R1,4(R3)
...
```
Forwarding

Efficient hardware solution to most data hazards
Forwarding

**Idea:** direct data access from intermediate registers

<table>
<thead>
<tr>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB R1,R3,R2</td>
</tr>
<tr>
<td>AND R2,R5,R12</td>
</tr>
<tr>
<td>OR R6,R2,R13</td>
</tr>
<tr>
<td>ADD R2,R2,R14</td>
</tr>
<tr>
<td>SW R15,100(R2)</td>
</tr>
</tbody>
</table>

**Czas (cykle zegara):**

<table>
<thead>
<tr>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rejestr R2</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
</tr>
<tr>
<td>Rejestr EX/MEM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Rejestr MEM/WB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-20</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Transfers (without Forwarding)

Only "forward" direction and final register modification
Transfers (with Forwarding)

- EX/MEM → ALU
- MEM/WB → ALU
Forwarding

- Hardware solution to most data hazards (between EX-MEM, EX-WB stages)
- Transfer of most-up-to-date results from Ex/Mem and Mem/WB to ALU input
- Hardware: combinatorial comparators of:
  - register numbers to be modified
    (Ex/Mem.Rd lub Mem/WB.Rd)
  - with
  - register numbers of operands for ALU
    (ID/Ex.Rs lub ID/Ex.Rt)
- Destination register is always updated in program order
Forwarding

Multiplexers at ALU input are controlled by forwarding alone (not by main control unit)

<table>
<thead>
<tr>
<th>Multiplexer</th>
<th>Dane</th>
<th>Opis</th>
</tr>
</thead>
<tbody>
<tr>
<td>For_A=00</td>
<td>ID/EX</td>
<td>Register File to ALU</td>
</tr>
<tr>
<td>For_A=10</td>
<td>EX/MEM</td>
<td>ALU to ALU</td>
</tr>
<tr>
<td>For_A=01</td>
<td>MEM/WB</td>
<td>MemData or ALU to ALU</td>
</tr>
<tr>
<td>For_B=00</td>
<td>ID/EX</td>
<td>Register File to ALU</td>
</tr>
<tr>
<td>For_B=10</td>
<td>EX/MEM</td>
<td>ALU to ALU</td>
</tr>
<tr>
<td>For_B=01</td>
<td>MEM/WB</td>
<td>MemData or ALU to ALU</td>
</tr>
</tbody>
</table>

Forwarding is transparent for control unit and does not increase its complexity.

For "deep" (or parallel) pipelines, forwarding complexity grows and limits its practical application.
Pipelined Architecture with Forwarding

(no jumps yet)
Forwarding in action (1)

- sub R1, R3, R2
- and R2, R5, R4
- or R4, R2, R4
- add R4, R2, R9
Forwarding in action (2)

- **add** $R4, R2, R9$
- **or** $R4, R2, R4$
- **and** $R2, R5, R4$
- **sub** $R1, R3, R2$

Diagram:

- Clock 4
- Instruction memory
- Forwarding unit
- Data memory
- ALU
- Registers
- Control
- Instruction
- IF/ID
- EX
- MEM
- WB
- MUX
- Forwarding unit

Before $<1>$
Forwarding in action (3)

sub R1, R3, R2
and R2, R5, R4
or R4, R2, R4
add R4, R2, R9

after<1>
add R4, R2, R9
or R4, R2, R4
and R2, R5, R4
sub R1, R3, R2

Clock 5
Forwarding in action (4)

- sub R1, R3, R2
- and R2, R5, R4
- or R4, R2, R4
- add R4, R2, R9
Forwarding – ALUSrc correction

- ALUSrc is set by main control unit only
- Autonomous forwarding operation require two independent multiplexers for ALU second input
"Hard" Data Hazards

- Forwarding cannot solve all data hazards
- *e.g.* Read After Write (RAW) – here: LW & ADD
"Hard" Data Hazards

Necessary pipeline stall

Diagram showing the program execution timeline and pipeline stages:

- LW R2,20(R1)
- AND R2,R5,R4
- OR R2,R6,R8
- ADD R4,R2,R9

Diagram illustrates the necessary pipeline stall due to data hazards.
Hardware Pipeline Stall

- Detection of hard data hazards must be done early (in ID)
- Additional RAW-hazard detection (combinatorial comparator) block is required in ID
- RAW-hazard detection block should be transparent for both main control and forwarding units

RAW-hazard detects:

- LW in stage EX (by examining ID/Ex.MemRead)
- Conflicting instruction in ID (by opcode: R-type, SW, BEQ)
- Matching numbers of registers:
  - ID/Ex.Rt (LW destination) and
  - IF/ID.Rs or IF/ID.Rt (conflicting instruction operands)
Hardware Pipeline Stall
Hardware Pipeline Stall in action (1)

\[ \text{lw} \quad R2, 20(R1) \]

and \[ R2, R5, R4 \]

or \[ R2, R4, R4 \]

add \[ R4, R2, R9 \]

Clock 2
Hardware Pipeline Stall in action (2)

lw R2, 20(R1)  
and R2, R5, R4  
or R2, R4, R4  
add R4, R2, R9

Clock 3
Hardware Pipeline Stall in action (3)

lw  R2, 20(R1)
and  R2, R5, R4
or  R2, R4, R4
add  R4, R2, R9

Clock 4
Hardware Pipeline Stall in action (4)

add R4, R2, R9

or R2, R4, R4

and R2, R5, R4

bubble

lw R2, 20(R1)

Clock 5
Hardware Pipeline Stall in action (5)

lw   R2, 20(R1)
and  R2, R5, R4
or   R2, R4, R4
add  R4, R2, R9

Clock 6
Hardware Pipeline Stall in action (6)

lw R2, 20(R1)
and R2, R5, R4
or R2, R4, R4
add R4, R2, R9

Clock 7
Control Hazard

- Any jump/branch breaks the natural sequence of instructions and spoils the pipeline (CPI > 1)
- Conditional branches (apart from address calculation) must also calculate the conditions – it may take time
- Jump/Branch execution will require a few following instructions to be invalidated

Effective solutions:
- Early Branch Detection – requires additional hardware
- Branch History Table – the best, but still based on guess
Control Hazard

- Late branch detection (our unmodified architecture):
  - branch condition evaluated at EX, active at MEM,
  - target instruction fetched after 3 cycles of delay

Program:

- 40 beq R1, R3, 7
- 44 and R2, R5, R12
- 48 and R6, R2, R13
- 52 add R2, R2, R14
- 72 lw R4, 50(R7)
Early Branch Detection Hardware
Early Branch Detection

- Condition (simple) is calculated in ID stage – only one stage of delay will be introduced (instruction in IF).
- Only simple condition is allowed (e.g. comparison), since the registers must be read from register file.
- Additional address needed in ID – dedicated for jump/branch address calculation.
- Instruction in IF must be invalidated – turned into NOP (effectively the same as invalidation).
- Invalidation in IF stage (→NOP) requires clearing the IF/ID intermediate register.
  - Providing, the NOP bit pattern (opcode + rest) is all 0's.
Early Branch Detection in action (1)

and R2, R5, R12

beq R1, R3, 7

sub R4, R8, R10

before <1>

before <2>
Early Branch Detection in action (2)

Iw R4, 50(R7)  

bubble (nop)  

beq R1, R3, 7  

sub R4, R8, R10  

before<1>
Branch History Table (BHT)

- BHT entry: recent branch instruction address & validated target address
  - **Tablica skoków** (*jump table*)

- (+) No need to use early detection hardware
- (+) Complex and late condition calculation is allowed
- (+) No processing delay at all
- (-) Target is still a guess and requires validation
- (-) Misprediction causes invalidation of many instructions
- (-) Complex prediction strategies are needed (hardware)