

Advanced Pipelining and Instruction Level Parallelism

Improving Performance Through Pipelining

- Key requirement: reduce stalls \Rightarrow keep the pipeline busy
- Need sufficient parallelism in programs
- Do programs have sufficient parallelism?
 - Example: gcc
 - * 17% control transfer: 5 instructions + 1 branch
 - * Beyond single block to get more instruction level parallelism

Conditions that Limit Parallelism

- (True) **Data dependencies** (RAW if a HW hazard)
 - 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*.
- **Anti-dependence** (WAR if a HW hazard)
 - Instruction *j* writes a register or memory location that instruction *i* reads from and instruction *i* is executed first
- **Output dependence** (WAW if a HW hazard)
 - Instruction *i* and instruction *j* write the same register or memory location; ordering between instructions must be preserved.

Control Dependence

- Example:

if p1 {S1;};	S1 is control dependent on p1
if p2 {S2;};	S2 is control dependent on p2, but not p1
- Two (obvious) constraints on code movement:
 - 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 order of exceptions and data flow.

Static vs. Dynamic Scheduling

- **Static scheduling:** compiler driven code movement and concurrency extraction
 - Complex and machine specific code optimizers
 - Limited in capability
- **Dynamic scheduling:**
 - Hardware-based concurrency extraction, performed at run-time
 - Simpler compiler and portable code
 - Complex hardware

Compile-time Techniques for Extracting Parallelism

- Example program fragment:

```
for (i=1, i<=1000, i++) {
  a[i] = a[i] + x;
}
```

- Assembly code from the program fragment:

```
Loop: LD    F0,0(R1)      ;F0 = vector element
      Stall
      ADDD  F4,F0,F2      ;add scalar in F2
      Stall
      Stall
      SD    0(R1),F4      ;store result
      SUBI  R1,R1,S       ;decrement pointer SB (DW)
      BNEZ  R1,LOOP      ;branch R1 != zero
      NOP                    ;delayed branch slot
```

Loop Unrolling

```

1 Loop: LD    F0,0(R1)
2      ADDD   F4,F0,F2
3      SD     0(R1),F4      ;drop SUBI & ENEZ
4      LD     F6,-8(R1)
5      ADDD   F8,F6,F2
6      SD     -8(R1),F8     ;drop SUBI & ENEZ
7      LD     F10,-16(R1)
8      ADDD   F12,F10,F2
9      SD     -16(R1),F12   ;drop SUBI & ENEZ
10     LD     F14,-24,(R1)
11     ADDD   F16,F14,F2
12     SD     -24(R1),F16
13     SUBI    R1,R1,#32     ;alter to 4*8
14     BNEZ   R1,LOOP
15     NOP

```

$15 + 4 \times (1 + 2) = 27$ clock cycles, or 6.8 per iteration.

Unrolled Loop that Minimizes Stalls

```

1 Loop: LD     F0,0(R1)
2      LD     F6,-8,(R1)
3      LD     F10,-16,(R1)
4      LD     F14,-24(R1)
5      ADDD   F4,F0,F2
6      ADDD   F8,F6,F2
7      ADDD   F12,F10,F2
8      ADDD   F16,F14,F2
9      SD     0(R1),F4
10     SD     -8(R1),F8
11     SD     -16(R1),F12
12     SUBI    R1,R1,#32
13     BNEZ   R1,LOOP
14     SD     S(R1),F16     ;8-32 = -24

```

14 clock cycles, or 3.5 per iteration

Determining Dependence

- Key questions:
 - What assumptions are made when moving code?
 - * OK to move store past SUBI even though changes register
 - * OK to move loads before stores: get right data?
 - * When is it safe for compiler to do such changes?
- Easy to determine for registers (fixed names)
- Hard for memory:
 - Does $100(R4) = 20(R6)$?
 - From different loop iterations, does $20(R6) = 20(R6)$?
 - Our example required compiler to know that if R1 doesn't change then:
 $0(R1) \neq -8(R1) \neq -16(R1) \neq -24(R1)$

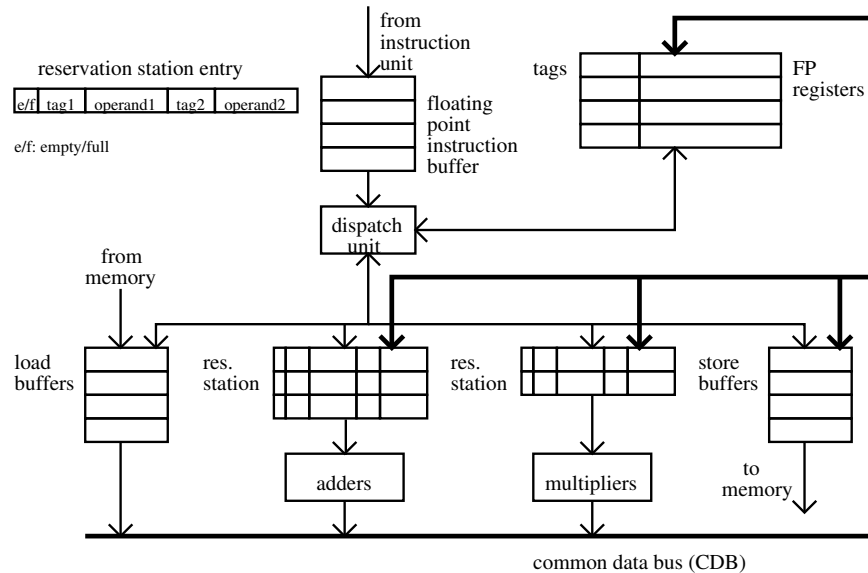
Dynamic Scheduling

- Key idea: Allow instructions behind stall to proceed

```
DIVD  F0,F2,F4
ADDD  F10,F0,F8
SUBD  F8,F8,F14
```

 - Enables out-of-order execution \Rightarrow out-of-order completion
 - ID stage checked both for structural and data dependencies
- Interesting case study: [Tomasulo's algorithm](#) implemented in the IBM 360/91
 - Distributed hazard detection/avoidance: Only structural hazards stall the pipeline, all other hazards delay only the affected instructions
 - Led to many techniques that are still in use (e.g., register renaming, data flow)

Architecture of IBM 360/91



Register File and Reservation Stations

- Each register has a **tag** indicating that either
 - the register is up to date
 - the register is out of date and the tag value is the id of the functional unit that will produce the up to date data
- Reservation Station**: a set of buffers where instructions are queued for a functional unit
 - An entry in a reservation station consists of the following:
 - * an empty/full bit
 - * the operands
 - * one tag per operand
 - An operand tag is either
 - * a special value indicating that the operand is “up to date”
 - * or the id of the functional unit that will produce the operand

Reservation Station Components

- OP – Operation to perform in the unit (e.g., + or –)
- Q_j, Q_k – Reservation stations producing source registers
- V_j, V_k – Value of source operands
- R_j, R_k – Flags indicating when V_j, V_k are ready
- Busy – Indicates reservation station and 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.

Execution

- All entries of all reservation stations continually monitor the common data bus (CDB)
 - if an operand is not up to date and the corresponding tag appears on the CDB, the corresponding data is “captured” and the operand becomes “up to date”
- A functional unit selects from its reservation station an operation with a complete set of ready operands (if any)
- After execution, a functional unit places on the CDB the result along with the id of the reservation station slot
- The reservation station slot is released (marked free)
- Instruction dispatch is stalled only on structural hazards
 - A structural hazards occurs when the reservation station of a needed functional unit is full

Dispatch: Load Operation

- A load instruction is placed in one of the load buffers
- The load buffer is marked busy and a load request is sent to memory
- The buffer id (tag) is associated with the load target register
 - the load target register is no longer “up to date”
 - if a later instruction needs that register, it is dispatched with a copy of the register tag
 - the later instruction waits until that tag appears on the CDB: that indicates that the load buffer has received the data
 - the later instruction gets its operand from the CDB
- When the data arrives from memory, it is placed (along with the tag) on the CDB and the load buffer is freed

RAW Hazards

- An instruction finds that at least one of its source registers is busy (computation in progress)
- The instruction is placed in the reservation station of the appropriate functional unit
 - a ready operand is fetched from the register file
 - an “in progress” operand receives the tag (from the register file) of the unit that will produce that operand
- The reservation station holds an operation until all operands are “up to date”
- All reservation station slots monitor the CDB for a matching tag
 - if a CDB tag matches that of a reservation slot, the corresponding data is “captured” and the operand is marked as “up to date”

WAW Hazards

- When an instruction is dispatched, the tag of the destination register is changed to the tag of the unit (reservation station slot) that receives the instruction
- Later instructions will get their data directly from that functional unit
- If a second instruction has the same register for destination, the tag is changed to the id of the functional unit that receives the second instruction
- The second instruction will no longer modify the register (since the tags won't match)

WAR Hazards

- If an instruction waits for an operand, it will wait for a functional unit to produce that operand
- If a later instruction needs to modify a register, it will not affect any waiting instructions (because no instruction waits for a register)

Tomasulo's Algorithm: Summary

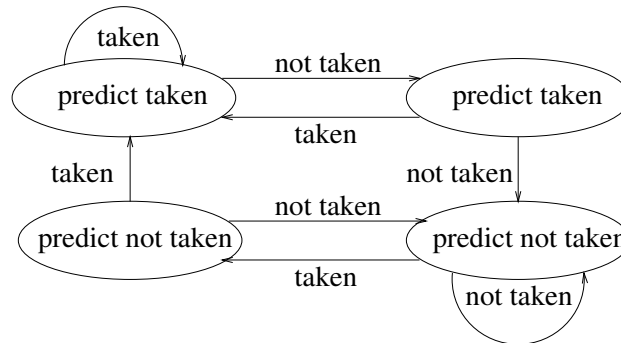
- Combines two key techniques:
 - Renaming of registers to a larger virtual set of registers
 - Buffering of source operands in the reservation stations
- **Advantage:** Eliminates performance degradation through anti- and output dependence (or WAR and WAW hazards); performance limited only by data (true) dependence
- **Drawbacks:** Complexity of hardware; single common data bus (CDB)
- **Caveat:** Performance improvement depends on the overhead imposed by control dependence

Reducing the Effect of Control Dependence

- Increase the size of basic blocks
 - Compile time techniques such as **Loop unrolling**, **trace scheduling**, etc.
- Branch prediction
 - **Static prediction:** prediction is made at compile time
 - **Dynamic prediction:** prediction is made at run time

Dynamic Branch Prediction

- Each conditional branch instruction has a few history bits associated with it. The bits are set at run time based on the branching history of the instruction, and are used to predict the outcome of the branch
- Example: Dynamic Branch Prediction using 2 information bits – a branch that strongly favors “taken” or “not taken” will be misrepresented only once. The two bits are used to encode the four states in the system.



Branch Target Buffer

- Branch prediction is useful in determining whether the branch is taken or not; but generally this is done in instruction decode stage
- Requires a single cycle stall!!!
- How to eliminate this stall? [Branch Target Buffer](#)
- Branch target buffer contains three fields:
 - Instruction address
 - Predicted PC address
 - Branch predicted taken/not-taken

Getting $CPI < 1$: Issuing Multiple Instructions/Cycle

- Two variations
- **Super-scalar**: Varying number of instructions/cycle (1 to 8), scheduled by compiler or by HW (Tomasulo)
 - IBM PowerPC, Sun SuperSparc, DEC Alpha, HP 7100
- **Very Long Instruction Words (VLIW)**: Fixed number of instructions (16) scheduled by the compiler
 - Merced architecture from Intel/HP

Limits to Multi-Issue Machines

- Inherent limitations of ILP
 - 1 branch in 5 instructions \Rightarrow how to keep a 5-way VLIW busy?
 - Latencies of units \Rightarrow many operations must be scheduled
 - Need about Pipeline Depth \times No. Functional Units of independent operations to keep machines busy
- Difficulties in building HW
 - Duplicate FUs to get parallel execution
 - Increase ports to register File
 - Increase ports to memory
 -