Advanced Pipelining

Dr. Yitzhak Birk
Electrical Engineering Department
Technion
Lecture outline

• Instruction-Level Parallelism (ILP).

• Hardware-based techniques to exploit ILP:
  – Superscalar processors.
  – Out-of-Order Execution.
  – Speculative Execution.

• Software-based techniques to exploit ILP:
  – Loop unrolling
  – SW pipelining.
  – Trace scheduling.
Data Dependences - Review

• **Types:**
  – True dependences (RAW): information needs to flow
  – Name dependences:
    » Antidependence (WAR)
    » Output dependence (WAW)
  – RAR is not a dependence

• **Complications:**
  – Values may pass through memory (mem disambiguation)
  – Dependence may depend on control flow (operand may potentially depend on more than one predecessor, depending on a branch)

• Name dependences can be solved with extra registers.
• Our goal: exploit parallelism by preserving program order only where it affects the outcome of the program.
Control Dependences - Review

• Simplistic Rules
  – An instruction that is control dependent on a branch cannot be moved before the branch.
  – An instruction that is not control dependent on a branch cannot be moved after the branch and become dependent on it.

• Insight: control dependence is not important by itself; instead, preserve:
  – Exception behavior
  – Data flow (actual flow of results to become operands etc.)

• Examples:
  – Moving LW before the branch appears OK, but what if it causes a memory protection exception?
  – Moving DSUBU before the branch appears a problem, but is OK if we know that its result is not used after skipnext.

DADDU R1, R2, R3
BEQZ R12, skipnext
DSUBU R4, R5, R6
LW R11, 0(R2)
DADDU R5, R4, R9
skipnext: OR R7, R8, R9
....
Dynamic Vs. Static Scheduling

• Two approaches to exploit ILP:
  – Static scheduling attempts to overcome limitations on ILP during compile-time using software techniques.
  – Dynamic scheduling attempts to overcome limitations on ILP dynamically by using hardware techniques at run-time.

• Static scheduling:
  – Compiler reorders instructions so as to separate dependent instructions.
  – The compiler has (in principle) unlimited look-ahead capability – analyzes the whole program during compile time.
  – Compiler attempts to enlarge basic block in order to reduce branch overhead and increase the number of candidates (instructions) for parallel execution.
  – Register renaming in software – map variables to machine registers.
  – At run time, hazard detection stalls problematic instruction and all following ones until the problem is resolved.
  – Cost:
    » complicated compiler and long compile time.
    » underutilized CPU resources (functional units)
Dynamic Vs. Static Scheduling

• Dynamic scheduling:
  – Whenever an instruction is not ready for execution (due to dependences), hardware tries to find other candidates for execution.
  – The look-ahead capability of the processor, also termed instruction window, is limited.
  – Instruction execution becomes out-of-order ⇒ instructions may complete out-of-order.
  – Branches may be predicted at run-time ⇒ control dependent instructions may be executed speculatively.
  – Virtual registers may be mapped to physical register at run-time.
  – Cost: complicated hardware.
  – Advantages:
    » Code compiled for one pipeline can run efficiently on another.
    » Can handle some cases in which dependencies are unknown at compile time.
    » Can use simpler compiler.

• See H&P3e Fig. 3.1 for a summary of techniques
Dynamic Scheduling

• Split instruction issue into two parts:
  – checking for structural hazards
  – waiting for the absence of a data hazard

• ID Pipeline stage broken into:
  – Issue: decode instruction and check for structural hazards
  – Read operands

• Instruction may proceed to EX even if previous instruction is still waiting for its operands.

• Result:
  – in-order issue
  – out-of-order execution and completion

• Complication: exceptions
  – Try to ensure that exception is only handled if instruction will definitely be executed
  – Problem when later instruction has already completed or earlier one has not
  – One solution: as part of speculative execution mechanism (H&P3e 3.7)
Managing Dyn. Sched: Tomasulo
Tomasulo’s Approach (IBM, 1967!!!)

- In-order issue into reservation stations
- In reservation station:
  - wait for operands (knowing which reservation station should produce each operand)
  - wait for functional unit to become available
- Results are sent from reservation station result buffer to registers and requesting instructions: broadcast over the common result (data) bus, CDB. Note: CDB may create structural hazards
- Load and Store buffers: similar to reservation stations
  - functional unit = memory port
  - operands: effective address (L+S), value (S)
Tomasulo – Instruction Issue

- Get next instruction from FIFO queue (in-order)
- If no reservation station, stall
- If there is a reservation station:
  - send instruction and available operands to reservation station
  - put names of producing reservation stations for missing operands
  - rename registers to eliminate WAR and WAW hazards.
Tomasulo - Execute

- Res. Sta. monitors CDB for any missing operand (handles RAW hazards)
- Once operands + functional unit are available, enter functional unit;
- If multiple instructions waiting on same execution unit:
  - FP: order generally does not matter
  - Load and store:
    - Load – Load: any order
    - Load – Store and Store – Store: maintain order at least through effective address calculation to help prevent hazards through memory (disambiguation)
- Exceptions:
  - do not allow execution to begin until all earlier branches have been resolved (so instruction is no longer speculative)
  - alternatively: allow execution to begin, but do not handle exceptions as long as instruction is speculative; delay it at entry to next stage.
Tomasulo – Write Results

- Write result + Reservation station ID on CDB, and from there to
  - registers
  - awaiting reservation stations
  - memory (Store)
Tomasulo – Control Data Structures

• Per reservation station:
  – status (busy)
  – per operand: available, or name of reservation station that will produce it
  – Op code

• Per Load/store buffer
  – effective address (initially the Immediate field of the instruction)
  – op code
  – value or name of RS producing it

• Per register in register file:
  – id of RS producing its next value
    (one per operand suffices because only new instructions need to learn the mapping; the others have stored it in the reservation station!)
Tomasulo - Remarks

• With branch prediction, Tomasulo’s approach permits parallel execution across basic blocks.
• Can be extended to multiple instruction issue per clock.
• Increased sensitivity to branch misprediction
• Key components:
  – dynamic scheduling
  – register renaming
  – dynamic memory disambiguation
• Invented 35 years ago, but presently in extensive use.
Multiple Instruction Issue

- **Goal:** break the CPI=1 limit
- **Approaches:**
  - **Very Long Instruction Word (VLIW)**
    - long “instruction packet” containing multiple operations
    - compiler takes full responsibility for correctness of parallelism
    - CPU only executes
    - inherently statically scheduled by the compiler
  - **Superscalar**
    - variable number of instructions issued per clock
    - CPU in charge of hazard detection and other coordination
    - static or dynamic scheduling
Each “long” instruction consists of multiple fixed-type instruction that specify what each ALU should do.

Machine control hardware is simple and relies on an intelligent compiler to fill the VLIW slots.

Instructions are sent to executions in-order, no-renaming, no speculation and other hardware-based trick.

If compiler does not find candidates for filling slots then it will assign a NOP instruction. ⇒ code density may increase.

Problem: dealing with backward compatibility when number of execution units increases.

Example of VLIW instruction format:

<table>
<thead>
<tr>
<th>ALU</th>
<th>ALU</th>
<th>Ld/st</th>
<th>FP</th>
<th>Branch</th>
</tr>
</thead>
</table>

VLIW (Very Long Instruction Word) Machines.
**Superscalar**

- Multiple pipelines running in parallel.
- Superpipelining – a pipeline with a relatively large number of stages (E.g., multiple EX substages)

![Superscalar Diagram]

### Superscalar

```plaintext
F
F
F
F
```

### Super-pipelined

```plaintext
F1 F2 D1 D2 E1 E2 WB1
```
Superscalar DLX

- Superscalar DLX: 2 instructions, 1 FP & 1 anything else
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - Need more ports for FP registers to do FP load & FP op in a pair

<table>
<thead>
<tr>
<th>Type</th>
<th>PipeStages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int. instruction</td>
<td>IF</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF</td>
</tr>
<tr>
<td>Int. instruction</td>
<td>ID</td>
</tr>
<tr>
<td>FP instruction</td>
<td>ID</td>
</tr>
<tr>
<td>Int. instruction</td>
<td>EX</td>
</tr>
<tr>
<td>FP instruction</td>
<td>EX</td>
</tr>
<tr>
<td>Int. instruction</td>
<td>MEM</td>
</tr>
<tr>
<td>FP instruction</td>
<td>MEM</td>
</tr>
<tr>
<td>Int. instruction</td>
<td>WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>WB</td>
</tr>
</tbody>
</table>

- Parallelism increases the impact of stall cycles, e.g. due to RAW data hazards. Example: 1 cycle load delay expands to 3 instructions in SS
  - Instruction in right half can’t use it, nor instructions in next slot
Modern Out-of-Order Processor Architecture

Instruction fetch (4 instructions) → Decoder → Multi-ported register-file

Instruction window (reservation stations)

Reorder buffer → EU #1 → EU #2 → EU #n
Statically Scheduled Superscalar

- Instructions are issued in order; all hazards checked for at issue time.
- Complication: Issue stage may dictate long clock cycle.
- Solution: pipeline the issue stage.
- Complication: need to check hazards with the other packet that is still in the issue stage
- Solution:
  - stage 1: decide how many instructions from the packet can issue
  - stage 2: check with the other packet (which has now finished Issue)
- Simplification: restrict number and/or type of instructions that can issue concurrently (e.g., one Int + one FP)
- Memory complication: packet crosses cache block boundary.
  Solutions:
  - restrict concurrent issue to same-cache-block instructions
  - use independent prefetch unit that loads FIFO, so problem goes away

- Even if HW cannot begin EX of more than one Int + one FP instr. per clock, dyn. sched. can fill reservation stations and loosen coupling, thereby increasing functional-unit utilization.

- Tomasulo simple extension: separate mechanism for FP and Int (low complexity); be careful with cases that involve both.

- Observation: key is assigning res. sta. and updating control tables.

- Approaches for arbitrary multiple-instr. issue:
  - half-clock per issue (two instr. per clock)
  - logic for handling two instructions at once
  - combine the two approaches (4 per clock)
Hardware-Based Speculation

• So far, speculative issue but non-speculative execution. Now, speculative execution.

• Components:
  – dynamic branch prediction to decide what to execute
  – speculative execution + capability to prevent permanent damage (prevent changes or be able to undo)
  – dynamic scheduling to deal with the scheduling of different combinations of basic blocks.

• Examples of use: most PowerPC, MIPS, Intel, Alpha and AMD.
Tomasulo-based HW Spec.

- Results must be forwarded speculatively among instructions to permit speculative execution, but
- Actual completion of an instruction may only take place once it is no longer speculative
- New stage: Instruction Commit
- New rule:
  - concurrent issue
  - out-of-order execution
  - in-order commit
Reorder Buffer

- One entry per issued instruction, in original order
- Fields of buffer entry:
  - Instruction type: branch / reg. / store
  - Destination (reg. number or memory address)
  - Value
  - Ready (execution completed; value ready)
- ROB completely replaces store buffers
- ROB replaces some of the functions (mainly holding the results) of the reservation stations, so ROB entry number is used to identify the source of an operand instead of the reservation station number.
Tomasulo with ROB – Instruction Issue

- Get next instruction from FIFO queue (in-order)
- If no reservation station or no ROB entry, stall
- If there is a reservation station and an ROB entry:
  - send instruction and available operands to reservation station
  - put names of producing ROB for missing operands in RS
  - put name of ROB entry in RS (so it can mark when broadcasting results on CDB)
  - (ROB is an alternative mechanism to register renaming).
Tomasulo with ROB - Execute

• Essentially unchanged by ROB
• Res. Sta. monitors CDB for any missing operand (handles RAW hazards)
• Once operands available, enter functional unit;
• If multiple instructions waiting on same execution unit:
  – FP: order generally does not matter
  – Load and store:
    » Load – Load: any order
    » Load – Store and Store – Store: maintain order at least through effective address calculation to help prevent hazards through memory (disambiguation)
• Exceptions:
  – do not allow execution to begin until all earlier branches have been resolved (so instruction is no longer speculative)
  – alternatively: allow execution to begin, but do not handle exceptions as long as instruction is speculative; delay it at entry to next stage.
Tomasulo with ROB – Write Results

• Write result + ROB ID on CDB, and from there to
  – ROB
  – awaiting reservation stations
    – DO NOT WRITE TO MEMORY OR REGISTERS
    – For store, write results only when both address and value are available

• Mark reservation station as free (but ROB entry is still in use)
Tomasulo with ROB - Commit

- Act when instruction reached head of ROB

- Store:
  - Update memory
  - Delete ROB entry

- Mispredicted branch:
  - Flush ROB
  - Restart execution at correct successor

- All other instructions:
  - Update registers
  - delete ROB entry
Tomasulo + ROB: Exceptions and Hazards

• Exceptions:
  – Exception is not handled until instruction is ready for commit (i.e., all previous ones have committed)
  – If a speculated instruction raises an exception, the exception is recorded in the ROB
  – If the speculation is incorrect, exception is flushed with the instruction.
  – Other, more aggressive methods are possible but more complicated

• Hazard through memory:
  – WAW and WAR are eliminated because updating of memory and registers occurs in order
  – RAW:
    » not allowing a Load to initiate the 2nd step of its execution if any active ROB entry occupied by a Store has a Destination field that matches the value of the A field of the Load
    » maintaining the program order for the computation of an effective address of a Load with respect to all earlier Stores.
Tomasulo – Control Data Structures

- **Per reservation station:**
  - status (busy)
  - per operand: available, or name of reservation station that will produce it
  - Op code

- **Per Load/store buffer**
  - effective address (initially the Immediate field of the instruction)
  - op code
  - value or name of RS producing it

- **Per register in register file:**
  - id of RS producing its next value
Tomasulo - Remarks

• With branch prediction, Tomasulo’s approach permits parallel execution across basic blocks.
• Can be extended to multiple instruction issue per clock.
• Increased sensitivity to branch misprediction
• Key components:
  – dynamic scheduling
  – register renaming
  – dynamic memory disambiguation

• Invented 35 years ago, but presently in extensive use.
Software-based techniques to exploit ILP
Loop-Level Parallelism

FP Loop: Where are the Hazards?

Loop:  
LD  F0,0(R1) ;F0=vector element  
ADDD F4,F0,F2 ;add scalar in F2  
SD  0(R1),F4 ;store result  
SUBI R1,R1,8 ;decrement pointer 8B (DW)  
BNEZ R1,Loop ;branch R1!=zero  
NOP ;delayed branch slot

<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>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>0</td>
</tr>
</tbody>
</table>
### FP Loop Hazards

Loop:

```
LD F0, 0(R1); F0 = vector element
ADDD F4, F0, F2; add scalar in F2
SD 0(R1); F4; store result
SUBIR1, R1, 8; decrement pointer 8B
(DW)
BNEZR1, Loop; branch R1 != zero
NOP; delayed branch slot
```

<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>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>0</td>
</tr>
</tbody>
</table>

- Where are the stalls?
FP Loop Showing Stalls

1. Loop: LD F0,0(R1); F0 = vector element
2. stall
3. ADDD F4, F0, F2; add scalar in F2
4. stall
5. stall
6. SD 0(R1), F4; store result
7. SUBI R1, R1, 8 ; decrement pointer 8B (DW)
8. BNEZ R1, Loop ; branch R1! = zero
9. stall ; delayed branch slot

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

• Rewrite code to minimize stalls?
Revised FP Loop Minimizing Stalls

1. Loop: LD F0,0(R1)
2. stall
3. ADDD F4,F0,F2
4. SUBI R1,R1,8
5. BNEZ R1,Loop ;delayed branch
6. SD 8(R1),F4 ;altered when move past SUBI

<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>
</tbody>
</table>

Unroll loop 4 times code to make faster?
Loop Unrolling
Unroll Loop Four Times

Rewrite loop to minimize stalls?

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration
Assumes R1 is multiple of 4
Unrolled Loop That Minimizes Stalls

• What assumptions made when moved 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 make such changes?

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 8(R1),F16 ; 8-32 = -24

14 clock cycles, or 3.5 per iteration
Compiler Perspectives on Code Movement

- Definitions: compiler concerned about dependences in program; whether or not there is a HW hazard depends on a given pipeline

- (True) Data dependences (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)?
Compiler Perspectives on Code Movement

- Another kind of dependence called **name dependence**: two instructions use the 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.
Compiler Perspectives on Code Movement

• Two (obvious) 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 order of exceptions and data flow
When can Multiple Iterations be Executed Concurrently?

- Example: Where are data dependencies? (A, B, C distinct & nonoverlapping)

```c
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]. This is a “loop-carried dependence”: between iterations

- Implies that iterations are dependent, and can’t be executed in parallel
**Loop Dependence - a Deeper Look**

- **Original loop:**  
  ```c
  for (i=1; i < 11; i++){
    A[i] = A[i] + B[i]; /* S1 */
    B[i+1] = C[i] + D[i]; /* S2 */
  }
  ```

- **Dependences**  
  - neither statement depends on itself (from a different iteration)
  - S1 depends on S2
  - S2 does not depend on S1

- **If there is no self-dependence or dependency cycle, LLP can be exploited**

- **Revised loop:**  
  ```c
  for (i=1; i < 10; i++){
    B[i+1] = C[i] + D[i]; /* S2 */
    A[i+1] = A[i+1] + B[i+1]; /* S1 */
  }
  ```

- **Dependences:**  
  - neither instruction depends on itself (from a different iteration)
  - no inter-iteration dependences
  - EXPOSED THE LLP

(LLP = Loop-Level Parallelism)
Loop Unrolling - Additional Remarks

- Requires additional registers
- Reduces the computational overhead
- When number of iterations unknown at compile time but known at loop entrance:
  - generate two versions of loop (original + unrolled)
  - for a loop with n iterations and unrolling k times:
    » run unrolled version \((n \div k)\) times
    » run original version \(\text{remainder } (n/k)\) times
Software Pipelining

- Observation: if operations different iterations of a loop are independent, then we can get ILP by taking instructions from different iterations.
- Software pipelining: reorganizes loops so that each iteration is made up of instructions chosen from different iterations of the original loop (- Tomasulo in SW)
**SW Pipelining Example**

**Before: Unrolled 3 times**

1. LD  F0,0(R1)
2. ADDD F4,F0,F2
3. SD  0(R1),F4
4. LD  F6,-8(R1)
5. ADDD F8,F6,F2
6. SD  -8(R1),F8
7. LD  F10,-16(R1)
8. ADDD F12,F10,F2
9. SD  -16(R1),F12
10. SUBI R1,R1,#24
11. BNEZ R1,LOOP

**After: Software Pipelined**

1. SD  0(R1),F4;   Stores M[i]
2. ADDD F4,F0,F2;  Adds to M[i-1]
3. LD  F0,-16(R1); loads M[i-2]
4. SUBI R1,R1,#8
5. BNEZ R1,LOOP
6. LD  F0,-8(R1)

**Pipelining Phases:**

- **IF:** Instruction Fetch
- **ID:** Instruction Decode
- **EX:** Execution
- **Mem:** Memory Access
- **WB:** Write Back

**Read F0:**
- ADDD
- LD

**Read F4:**
- LD

**Write F4:**
- ADDD
- SD

**Write F0:**
- BNEZ
- SD
SW Pipelining Vs. Loop Unrolling

- Same number of regs as in original program
- Same overhead as in original program
- (Almost) same code length as in original program
- More registers than in original program
- Lower overhead
- Longer code

When functional units have long latencies, we may have to separate generation from consumption by several iterations. Can combine the two techniques.