Technical University of Cluj-Napoca
Computer Science Department

Computer Architecture

Lecturer: Mihai Negru

2nd Year, Computer Science

Lecture 9: Advanced Pipelining

http://users.utcluj.ro/~negrum/

Data Hazard Classification

- Data hazards: depend on the order of read and write accesses to the registers from the Register File.

- Consider two instructions i and j, with i occurring before j.

- Possible data Hazards:

  - RAW – read after write
    - i: R2 ← R1 + R3
    - j: R4 ← R2 + R3
    - j tries to read R2 before i writes it, so j incorrectly gets the old value for R2 .
    - The most common type of hazard → forwarding to overcome it

Data Hazard Classification

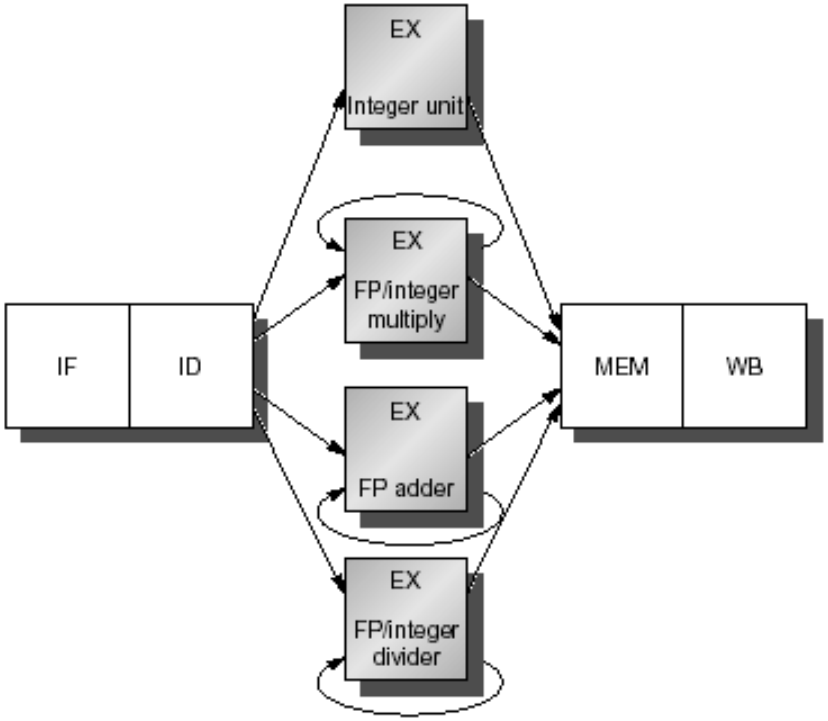
- Data hazards:
  - WAW – write after write
    - i: R2 ← R4 x R7
    - j: R2 ← R1 + R3
    - *j* tries to write R2 before it is written by *i*.
    - Writes in wrong order, leaving the value written by i rather than the value written by j in the destination register.
    - This hazard is present only in pipelines that write in more than one pipe stage (or allow an instruction to proceed even when a previous instruction is stalled).
    - Can't happen in MIPS 5 stage pipeline because:
      - All instructions take 5 stages, and
      - Writes are always in stage 5 (WB)

Data Hazard Classification

- Data hazards:
  - WAR – write after read
    - i: R1 ←R2 + R3
    - j: R3 ← R4 + R5
    - *j* tries to write a destination before it is read by *i*, so *i* incorrectly gets the new value – instruction i is stuck
    - Can't happen in MIPS 5 stage pipeline because:
      - All instructions take 5 stages
      - Reads are always in stage 2 (ID) and
      - Writes are always in stage 5 (WB)
    - This hazard occurs when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source late in the pipeline.
  - RAR – read after read is not a hazard!

Advanced Pipelining

- Pipeline MIPS with variable length multi-cycle operations
  - Extended MIPS pipeline to handle Floating Point (FP) operations
  - The FP operations can have different latencies for variable length operations
  - Latency – number of clock cycles necessary to execute the operation

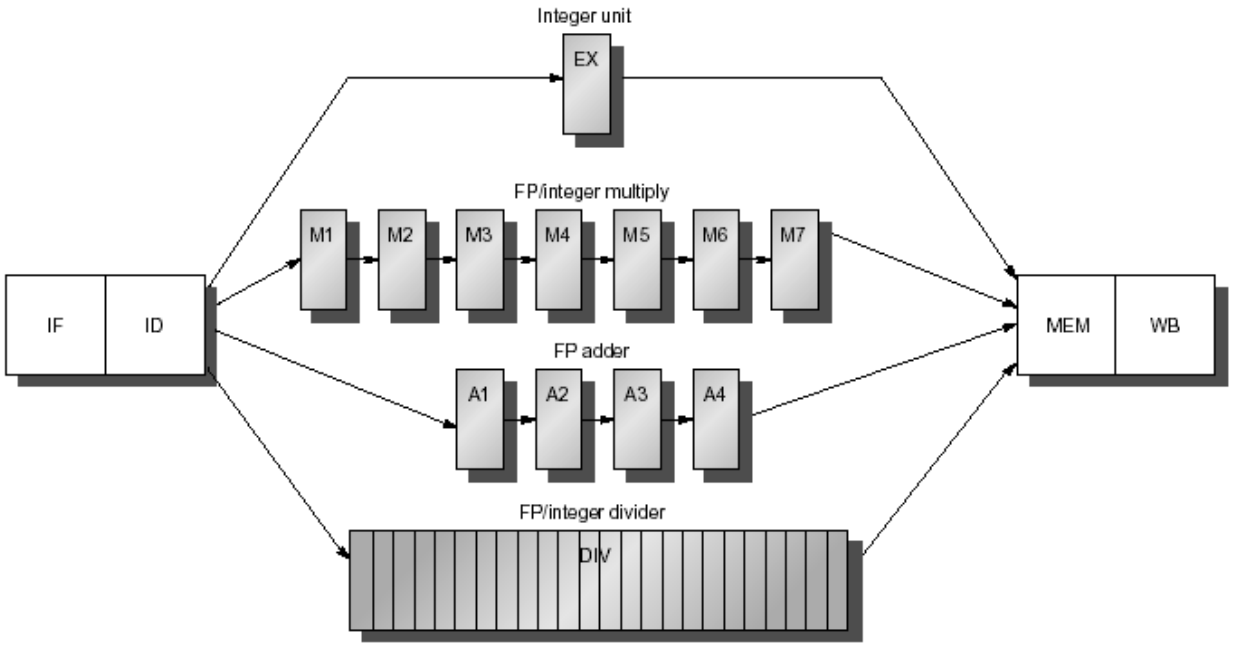


MIPS Pipeline with 4 Functional Units: 1 – INT, 3 – FP

Advanced Pipelining

- EX Integer unit handles loads and stores, integer ALU operations, and branches;
- EX FP/Integer multiply, FP adder, FP/Integer divider.
- Only one instruction could be issued in a clock cycle,
- All instructions go through the standard pipeline stages:
  - IF, ID, EX, MEM, WB.
- The FP operations are executed in several clock cycles in EX.
- After EX stage → MEM and WB to complete execution.
- If the FP units are not pipelined
  - No new instruction may issue until the previous one leaves the EX stage.
- If an instruction cannot proceed to the EX stage, the entire pipeline behind that instruction will be stalled.

Advanced Pipelining



Pipeline in pipeline: A pipeline that supports multiple outstanding FP operations

- Latency: the number of intervening cycles between an instruction that produces a result and an instruction that uses the result.
- Initiation or repeat interval: the number of cycles that must elapse between issuing two operations of a given type

The latency of the FP operations increases the frequency of RAW hazards and stalls!

Advanced Pipelining

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

- The FP **multiplier** and **adder** are fully pipelined: **7** and **4** stages, respectively.
- The FP **divider** is not pipelined; it requires **24** clock cycles to complete.
- The latency in instructions between the issue of a FP operation and the use of the result of that operation without incurring a RAW stall is determined by the number of cycles spent in the execution stages.
- Example: the fourth instruction after a FP add can use the result of the FP add.
- For integer ALU operations, the depth of the execution pipeline is always one and the next instruction can use the results.
- Both FP loads and Integer loads complete during MEM, which means that the memory system must provide either 32 or 64 bits in a single clock cycle.

Advanced Pipelining

MULD	IF	ID	**M1**	M2	M3	M4	M5	M6	**M7**	MEM	WB
ADDD		IF	ID	**A1**	A2	A3	**A4**	MEM	WB		
LD			IF	ID	**EX**	**MEM**	WB				
SD				IF	ID	**EX**	MEM	WB			

The pipeline timing diagram for a set of independent FP operations.

- The stages where data is needed
- The stages where a result is available.
- FP loads and stores use a 64-bit path to memory, so the pipelining timing is just like an integer load or store.
- The structure of the pipeline requires the introduction of additional pipeline registers (e.g., A1/A2, A2/A3, A3/A4).
- The ID/EX register must be expanded to connect ID to EX, DIV, M1, and A1
- The forwarding is similar – check if the destination register in any of EX/MEM, A4/MEM, M7/MEM, D/MEM, or MEM/WB registers is one of the source registers of a FP instruction.
  - If so, the appropriate input multiplexer will have to be enabled so as to choose the forwarded data.

Advanced Pipelining – Problems

- The division unit is not fully pipelined, structural hazards can occur.
  - Should be detected and issuing instructions will be stalled.
- The instructions have varying running times, the number of register writes in a cycle can be larger than 1.
- WAW hazards are possible, since instructions no longer reach WB in order.
- WAR hazards are not possible, the register reads always occur in ID.
- Instructions can complete in a different order than they were issued, causing problems with exceptions.
- Because of longer latency of operations, stalls for RAW hazards will be more frequent.

Advanced Pipelining

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD F4, 0 (R2)	IF	ID	EX	MEM	WB												
MULD F0, F4, F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADDD F2, F0, F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	WB
SD 0 (R2), F2					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM

- Each instruction in this sequence is dependent on the previous and proceeds as soon as data is available
  - assumes the pipeline has full forwarding.

- The SD must be stalled an extra cycle so that its MEM does not conflict with the ADDD.
  - Extra hardware can handle this case (write buffer).

Dynamic Scheduling – Scoreboard Method

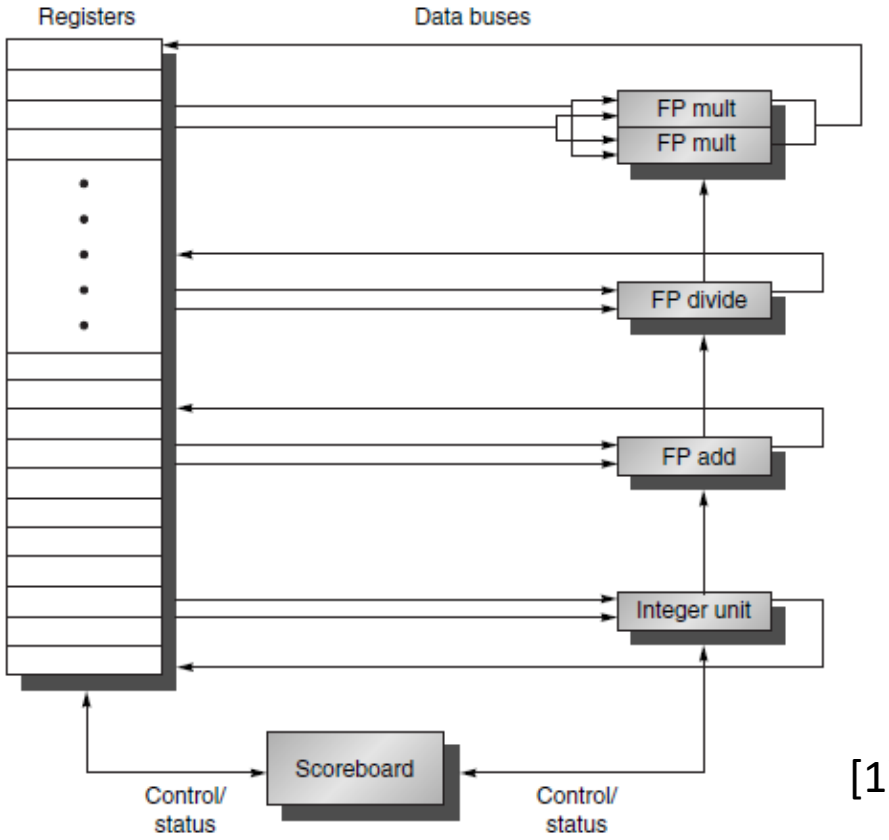
- Hardware scheme – instruction parallelism
  - Why in HW at run time?
    - Works when one does not know real data dependence at compile time
    - Code is not machine dependent
    - Simpler compiler
  - Key idea: Allow instructions behind stall to proceed.

    DIVD F0, F2, F4

    ADDD F10, F0, F8

    SUBD F12, F8, F14
- Enables out-of-order execution → out-of-order completion
- Split ID stage to check both for structural & data dependencies
- Scoreboard realized in 1963 for CDC 6600 (First supercomputer)
- No forwarding!
- Instructions execute when they are not dependent on previous instructions and there are no hazards.

Dynamic Scheduling – Scoreboard Method



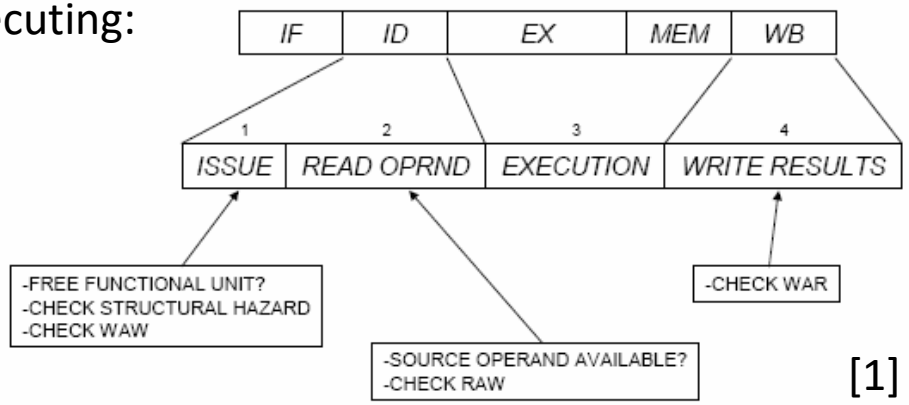
[1]

The basic structure of a MIPS processor with a Scoreboard

- The scoreboard's function is to control instruction execution (vertical control lines).
- All data flows between the RF and the FUs over the buses (the horizontal lines).
- 2 FP multipliers, 1 FP divider, 1 FP adder, and 1 integer unit.
- 2 inputs busses and 1 output bus serve a group of FUs.

Dynamic Scheduling – Scoreboard Method

- Every instruction goes through the scoreboard, where a record of the data dependences is constructed
  - this step corresponds to instruction issue and replaces part of the ID step in the MIPS pipeline.
- The scoreboard then determines when the instruction can read its operands and begin execution.
- If the scoreboard decides the instruction cannot execute immediately, it monitors every change in the hardware and decides when the instruction can execute.
- The scoreboard also controls when an instruction can write its result into the destination register.
- Thus, all hazard detection and resolution is centralized in the scoreboard.
- Each instruction undergoes four steps in executing:
  - Issue
  - Read operands
  - Execution
  - Write results



[1]

Dynamic Scheduling – Scoreboard Method

- Scoreboard stages:
  - Issue – decode instructions & check for structural & WAW hazards (ID1)
    - If a FU for the current instruction is free and no other active instruction has the same destination register (WAW) → The scoreboard issues the instruction to the FU and updates its internal data structure.
    - If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared
  - Read operands – wait until no RAW data hazards, then read operands (ID2)
    - A source operand is available if
      - No earlier issued active instruction is going to write it, or if
      - The register containing the operand is being written by a currently active FU.
    - Registers are only read when they are both available.
    - When the source operands are available, the scoreboard tells the FU to proceed to read the operands from the registers and begin execution.
    - Because the operands are read only when both are available in the Register File, the scoreboard does not take advantage of forwarding!
    - The scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution out of order

Dynamic Scheduling – Scoreboard Method

- Scoreboard stages:
  - Execute – operations with the operands (EX)
    - The FU begins execution upon receiving the operands.
    - When the result is ready, it notifies the scoreboard that it has completed execution.
  - Write result – finish execution (WB)
    - Once the scoreboard is aware that the FU has completed execution, the scoreboard checks for WAR hazards.
    - If none, it writes results.
    - If WAR, then it stalls the instruction
    - Example

      DIVD F0, F2, F4

      ADDD F10, F0, F8

      SUBD F8, F8, F14
    - CDC 6600 scoreboard would stall SUBD until ADDD reads operands

  - **Scoreboard → In-order issue – out-of-order execute & commit**

Dynamic Scheduling – Scoreboard Method

- Applying the Scoreboard method
  - Timing: Latency (clock cycles) FP Addition = 2, Multiplication = 10, Division = 40, Read Op = 1, Execute Load = 1
  - In order Issue
  - Stage and Hazard test
    - Issue (Iss): Structural, WAW
    - Read operand (Rop): RAW
    - Write result (Wr): WAR
  - For the given instruction sequence:
    - Complete the following table

#	Instruction	Str	WAW	Iss	RAW	Rop	stE	endE	WAR	Wr
1	**LD F6 34+R2**									
2	**LD F2 45+R3**									
3	**MULD F0 F2 F4**									
4	**SUBD F8 F6 F2**									
5	**DIVD F10 F0 F6**									
6	**ADDD F6 F8 F2**									

Dynamic Scheduling – Scoreboard Method

- Applying the Scoreboard method
  - stE: start Execution
  - endE: end Execution
  - Str (structural), WAW, RAW, WAR: hazard type
  - the number in hazard column shows the related instruction's number
  - The number in the Iss, Rop, stE, endE and Wr columns – clock cycle number

#	Instruction	Str	WAW	Iss	RAW	Rop	stE	endE	WAR	Wr
1	**LD F6 34+R2**			1		2	3	3		4
2	**LD F2 45+R3**	1		5		6	7	7		8
3	**MULD F0 F2 F4**			6	2	9	10	19		20
4	**SUBD F8 F6 F2**			7	2	9	10	11		12
5	**DIVD F10 F0 F6**			8	3	21	22	61		62
6	**ADDD F6 F8 F2**	4		13		14	15	16	5	22

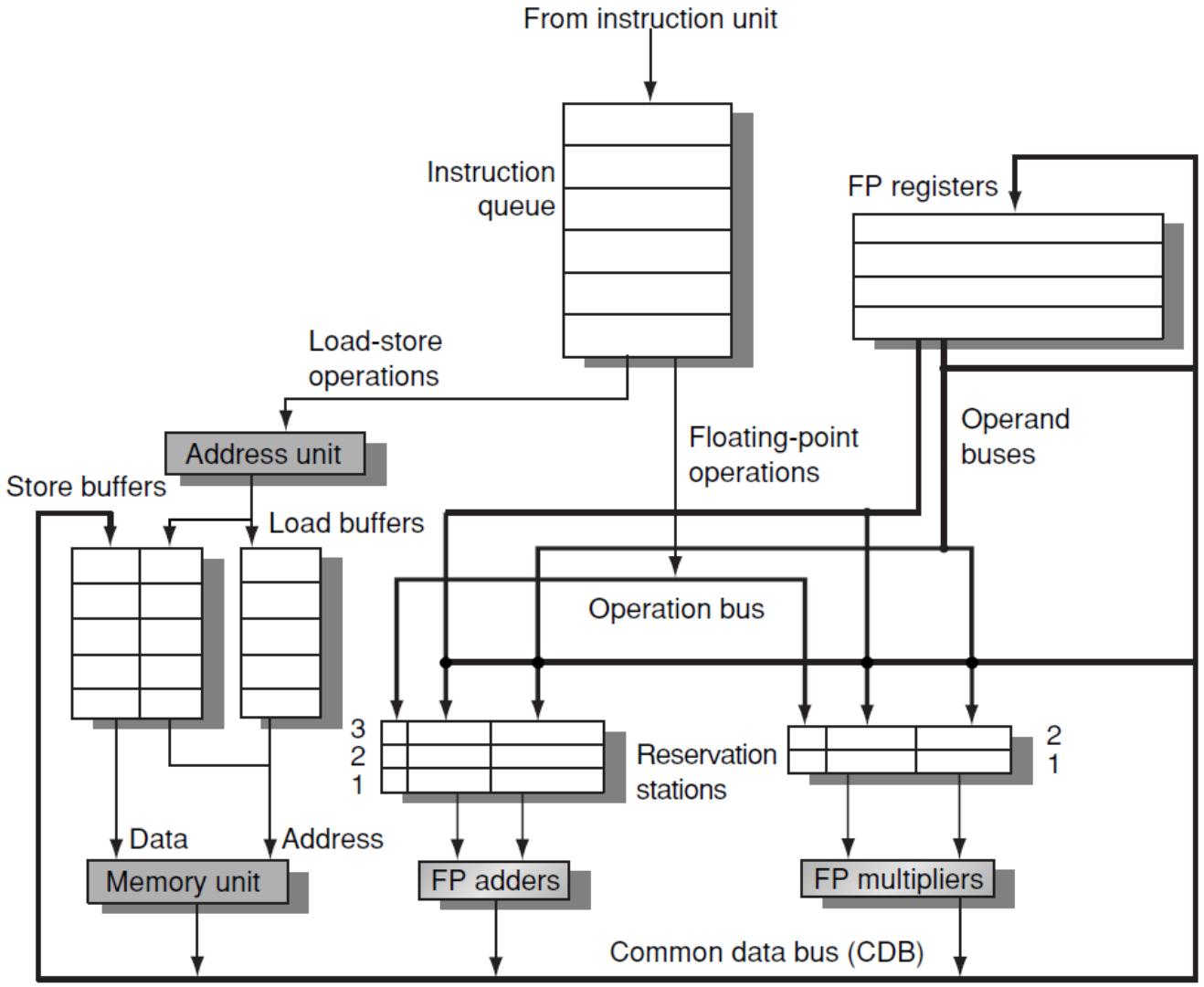
Dynamic Scheduling – Scoreboard Method

- Scoreboard limits:
  - A scoreboard uses the available ILP to minimize the number of stalls arising from the program's true data dependences.
  - In eliminating stalls, a scoreboard is limited by several factors:
    - The amount of parallelism available among the instructions
      - If each instruction depends on its predecessor, no dynamic scheduling scheme can reduce stalls
    - The number of scoreboard entries
      - Determines how far ahead the pipeline can look for independent instructions. The set of instructions examined as candidates for potential execution is called the window. The size of the scoreboard determines the size of the window. We assume a window does not extend beyond a branch → only straight-line code from a single basic block is inside the scoreboard
    - The number and types of functional units
      - Determines the importance of structural hazards, which can increase when dynamic scheduling is used
    - The presence of anti-dependences and output dependences
      - These lead to WAR and WAW stalls
    - No forwarding hardware

Dynamic Scheduling – Tomasulo Method

- Tomasulo algorithm
  - for IBM 360/91 about 3 years after CDC 6600 (1966)
  - Goal: High Performance dynamic scheduling of the execution without special compilers
  - Differences between IBM 360 & CDC 6600 ISA
  - IBM has only 2 register specifiers / instr. vs. 3 in CDC 6600
  - IBM has 4 FP registers vs. 8 in CDC 6600
  - Why Study? Implemented in
    - Alpha 21264
    - HP 8000
    - MIPS 10000
    - Pentium II
    - PowerPC 604
    - …

Dynamic Scheduling – Tomasulo Method



[1]

The basic structure of MIPS FP Unit using Tomasulo's algorithm

Dynamic Scheduling – Tomasulo Method

- Tomasulo vs. Scoreboard
  - Control & buffers distributed with Function Units (FU) vs. centralized in scoreboard
  - FU buffers called "Reservation Stations" (RS); buffer the operands of instructions waiting to issue
    - RS fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from the Register File
    - Pending instructions designate the RS that will provide their input
    - When successive writes to a register appear, only the last one is actually used to update the register
  - Registers in instructions replaced by values or pointers to RS; called **register renaming**
    - Avoids WAR, WAW hazards
    - More reservation stations than registers so can do optimizations that compilers cannot.
  - Results are broadcasted over Common Data Bus (CDB)
  - Load and Stores treated as FUs with RSs as well
  - Common data bus: data + source
    - 64 bits of data + 4 bits of Functional Unit source address
    - Write if matches → expected Functional Unit produces result
    - Does the broadcast

Dynamic Scheduling – Tomasulo Method

- Floating-point operations are sent from the instruction fetch unit into a queue.
- The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards.
- Load buffers to hold the addresses for memory reads that are waiting for the CDB.
- Similarly, store buffers are used to hold the destination memory addresses of stores waiting for their operands.
- All results from either the FP units or the load unit are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers.
- The FP adders implement addition and subtraction, while the FP multipliers do multiplication and division

Dynamic Scheduling – Tomasulo Method

- Tomasulo stages:
  - Issue – get an instruction from the floating-point operation queue.
    - If the operation is a floating-point operation, issue if there is an empty reservation station, and send the operands to the reservation station if they are in the registers.
    - If the operation is a load or store, it can issue if there is an available buffer.
    - If there is no empty reservation station or an empty buffer, then there is a structural hazard and the instruction stalls until a station or buffer is freed.
    - This step also performs the process of renaming registers.
  - Execute – if one or more operands is not yet available, monitor the CDB
    - When an operand becomes available, it is placed into the corresponding reservation station.
    - When both operands are available, execution starts.
    - This step checks for RAW hazards
  - Write result – finish execution (WB)
    - When the result is available, write it on the CDB and from there into: registers, reservation stations waiting for the result, and to any waiting store buffers.
    - Mark the used reservation station as available

Dynamic Scheduling – Tomasulo Method

- The major advantages of the Tomasulo scheme are:
  - The distribution of the hazard detection logic
  - The elimination of stalls for WAW and WAR hazards.
  - The reduction in structural hazards – more reservation stations than functional units
  - CDB conflict may appear
  - The first advantage arises from the distributed reservation stations and the use of the CDB.
    - If multiple instructions are waiting on a single result, and each instruction already has its other operand, then the instructions can be released simultaneously by the broadcast on the CDB.
    - In Scoreboard the results are written in registers, and then read by the waiting instructions
  - WAW and WAR hazards are eliminated by renaming registers using the reservation stations, and by storing operands into the reservation station as soon as they are available.

Dynamic Scheduling – Tomasulo Method

- Applying the Tomasulo Method
  - Timing: Latency (clock cycles) FP Addition = 2, Multiplication = 10, Division = 40, Read Op = 1, Execute Load = 1
  - In order Issue
  - Stage and Hazard test
    - Issue (Iss): Structural
    - Execute: RAW
    - Write result (Wr): CDB conflict
  - For the given instruction sequence:
    - Complete the following table

#	Instruction	Str	Iss	RAW	stE	endE	CDB	Wr
1	**LD F6 34+ R2**							
2	**LD F2 45+ R3**							
3	**MULD F0 F2 F4**							
4	**SUBD F8 F6 F2**							
5	**DIVD F10 F0 F6**							
6	**ADDD F6 F8 F2**							

Dynamic Scheduling – Tomasulo Method

- Applying the Tomasulo Method
  - stE: start Execution
  - endE: end Execution
  - Str (structural), RAW: hazard type
  - CDB – common data bus conflict
  - the number in hazard column shows the related instruction's number
  - The number in the Iss, stE, endE and Wr columns – clock cycle number

#	Instruction	Str	Iss	RAW	stE	endE	CDB	Wr
1	**LD F6 34+ R2**		1		2	3		4
2	**LD F2 45+ R3**		2		3	4		5
3	**MULD F0 F2 F4**		3	2	6	15		16
4	**SUBD F8 F6 F2**		4	2	6	7		8
5	**DIVD F10 F0 F6**		5	3	17	56		57
6	**ADDD F6 F8 F2**		6	4	9	10		11

Dynamic Scheduling – Tomasulo Method

- Tomasulo Drawbacks
  - Complexity
  - Many associative stores (CDB) at high speed
  - Performance limited by Common Data Bus
    - Multiple CDBs → more FU logic for parallel stores
- Load/store disambiguation
  - The store buffers memorize the addresses and data such that the CPU does not wait for memory writes
  - The load and store can safely be done in a different order, provided the load and store access different addresses.
  - For every load the store addresses from the store buffers are examined
  - If a store address matches the load address, we must stop and wait until the store buffer gets a value; we can then access it or get the value from memory.

Dynamic Scheduling – Tomasulo Method

- Tomasulo Summary
  - Reservations stations: renaming to larger set of registers + buffering source operands
    - Prevents number of registers as bottleneck
    - Avoids WAR, WAW hazards of Scoreboard
    - Allows loop unrolling in HW
  - Contributions employed in modern processors
    - Dynamic scheduling
    - Register renaming
    - Load/store disambiguation
  - Tomasulo's scheme is appealing if one needs to pipeline an architecture for which it is difficult to schedule code or that has a shortage of registers
  - The adv. of the Tomasulo approach vs. compiler scheduling for an efficient single-issue pipeline are probably fewer than the costs of implementation.
  - But, due to the simultaneous issuing of more instructions (superscalar) and the necessity to improve the performance of difficult-to schedule code (static), **register renaming and dynamic scheduling are applied in modern processors.**

ILP: Loop Unrolling and Software Pipelining

- ILP: Overlap execution of unrelated instructions
  - One opportunity – loop level parallelism
  - The simplest and most common way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations of a loop.
  - This type of parallelism is often called *loop-level parallelism*
- Basic Pipeline Scheduling and Loop Unrolling
  - Suppose standard MIPS pipeline with branch resolved in ID stage
  - No structural hazard, at each clock cycle a new instruction is issued

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

Latency column shows the number of clock cycles needed to avoid a stall

ILP: Loop Unrolling and Software Pipelining

- Example – add the same value to the elements of an array in memory

```
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 8 bytes (DWord)
 BNEZ    R1, Loop        branch R1 != zero
 NOP                     branch delay slot
```

FP Loop: Where are the Hazards?

- Without any scheduling the loop will execute as follows:

```
     Loop:
1    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 8 bytes (DW)
8    BNEZ    R1, Loop       branch R1 != zero
9    stall                  branch delay slot
```

FP loop with stalls

- 9 clock cycles
- Rewrite the code to minimize the stalls

ILP: Loop Unrolling and Software Pipelining

```
     Loop:
1    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     why 8(R1): to compensate the effect of SUBI
                          when accessing the memory location
```

- Replace BNEZ stall with SD and change address of SD
- 6 clocks: Unroll loop 4 times code to make faster?

ILP: Loop Unrolling and Software Pipelining

Unroll loop 4 times

```
      Loop:
1     LD    F0, 0(R1)      after LD 1 cycle stall
2     ADDD  F4, F0, F2     after ADDD 2 cycles stall
3     SD    0(R1), F4      drop SUBI & BNEZ
4     LD    F6, -8(R1)
5     ADDD  F8, F6, F2
6     SD    -8(R1), F8     drop SUBI & BNEZ
7     LD    F10, -16(R1)
8     ADDD  F12, F10, F2
9     SD    -16(R1), F12   drop SUBI & BNEZ
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
```

- Stalls:
  - LD – 1 stall
  - ADDD – 2 stalls
- 15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration
- Assumes R1 is multiple of 4
- How many registers do we need? 3 vs. 9

- This unrolled version is slower than the scheduled version of the original loop.

ILP: Loop Unrolling and Software Pipelining

Unrolled loop that minimizes stalls

```
      Loop:
1     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 the SUBI effect
```

- What assumptions made when code moved?
- OK to move store past SUBI even though changes register
- OK to move loads before stores: get right data?
- 14 clock cycles, or 3.5 per iteration.

- Using new registers is equivalent to register renaming (done by programmer or compiler)

Problems – Homework

- A MIPS machine has the following resources:

Functional Unit	Nr.	Clock Cycles in EX Stage
Integer Unit	1	1
FADD Unit	2	3
FMUL Unit	1	10
FDIV Unit	1	40

- Show the execution of the following instructions with
  - Scoreboard Method
  - Tomasulo Method
  - Draw the table for each method
  - Highlight the possible hazards and how they are resolved

#	Instruction
1	ADDD F6,F8,F2
2	SD F6, 100(R3)
3	MULD F6,F6,F2
4	ADDD F6,F6,F4
5	DIVD F8,F1,F6
6	ADDD F6,F8,F2
7	ADDD F1,F3,F2

References

1. D. A. Patterson and J. L. Hennessy, “Computer Organization and Design: A Quantitative Approach”, 5th edition, ed. Morgan-Kaufmann, 2011
2. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
3. MIPS32™ Architecture for Programmers, Volume I: “Introduction to the MIPS32™ Architecture”.
4. MIPS32™ Architecture for Programmers Volume II: “The MIPS32™ Instruction Set”.