Advanced Pipelining and ILP – Part 1

What is advanced pipelining?

- high-performance float-point pipeline using
  - static pipeline scheduling and loop unrolling
  - dynamic scheduling with scoreboard
  - dynamic scheduling register renaming
  - dynamic branch prediction
  - multiple-instruction issuing
  - software pipelining and trace scheduling
  - speculation
  - dynamic memory disambiguation

  to reduce or eliminate
  - control stalls
  - RAW, WAW and WAR stalls

  exploiting more instruction level or loop level parallelism

Instruction Level Parallelism (ILP)

Instruction Level Parallelism (ILP)

- the parallelism in the instruction sequence that allows the instruction executions to be overlapped in the pipeline.
- ILP is a property of programs and closed related to data dependences in the programs.
- The purpose of advanced FP pipelining is to
  - eliminate WAW and WAR dependences
  - reduce the stalls caused by RAW dependences
  - reduces the stall caused by control dependences

  using various compiler and pipeline implementation techniques.

Static Pipeline Scheduling and Loop Unrolling – Example

Latencies between instructions in the MIPS FP pipeline

Instruction producing result	Instruction using result	Latency
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

- the basic information for the compiler pipeline scheduling to reduce RAW stalls

Example

```
for (i=1000; i>0; i--)
    x[i] = x[i] + s
```

```
loop: LD     F0, 0(R1)
      ADDD   F4, F0, F2
      SD     0(R1), F4
      SUBI   R1, R1, #8
      BEQZ   R1, loop
```

Actual Performance with RAW and Control Stalls

```
                                 clock cycle issued
loop: LD     F0, 0(R1)                 1
      stall                            2
      ADDD   F4, F0, F2                3
      stall                            4
      stall                            5
      SD     0(R1), F4                 6
      SUBI   R1, R1, #8                7
      stall                            8
      BEQZ   R1, loop                  9
      stall                           10
```

Pipeline scheduling can reduce it to 6 cycles per iteration

```
                                     clock cycle issued
loop: LD     F0, 0(R1)                     1
      SUBI   R1, R1, #8                    2
      ADDD   F4, F0, F2                    3
      stall                                4
      BEQZ   R1, loop                      5
      SD     8(R1), F4                     6
```

How is it done?

- dependence analysis to find flow (RAW) dependences through registers
- use latency to separate dependent instructions
- fill stalls with independent instructions

Loop Unrolling

- assume that the loop with n iterations execute at least k iterations
- unroll every k iterations to form a larger body, thus reducing the number of branches

```
loop: LD     F0, 0(R1)
      ADDD   F4, F0, F2
      SD     0(R1), F4
      SUBI   R1, R1, #8
      LD     F0, 0(R1)
      ADDD   F4, F0, F2
      SD     0(R1), F4
      SUBI   R1, R1, #8
      LD     F0, 0(R1)
      ADDD   F4, F0, F2
      SD     0(R1), F4
      SUBI   R1, R1, #8
      LD     F0, 0(R1)
      ADDD   F4, F0, F2
      SD     0(R1), F4
      SUBI   R1, R1, #8
      BEQZ   R1, loop
```

Remove flow dependences through symbolic analysis of the value of R1.

```
loop: LD     F0, 0(R1)
      ADDD   F4, F0, F2
      SD     0(R1), F4
      LD     F0, -8(R1)
      ADDD   F4, F0, F2
      SD     -8(R1), F4
      LD     F0, -16(R1)
      ADDD   F4, F0, F2
      SD     -16(R1), F4
      LD     F0, -24(R1)
      ADDD   F4, F0, F2
      SD     -24(R1), F4
      SUBI   R1, R1, #32
      BEQZ   R1, loop
```

Remove anti-dependence (WAR) and output dependence (WAW) through register renaming.

```
loop: LD     F0, 0(R1)
      ADDD   F4, F0, F2
      SD     0(R1), F4
      LD     F6, -8(R1)
      ADDD   F8, F6, F2
      SD     -8(R1), F8
      LD     F10, -16(R1)
      ADDD   F12, F10, F2
      SD     -16(R1), F12
      LD     F14, -24(R1)
      ADDD   F16, F14, F2
      SD     -24(R1), F16
      SUBI   R1, R1, #32
      BEQZ   R1, loop
```

Do pipeline scheduling again.

```
loop: LD     F0, 0(R1)
      LD     F6, -8(R1)
      LD     F10, -16(R1)
      LD     F14, -24(R1)
      ADDD   F4, F0, F2
      ADDD   F8, F6, F2
      ADDD   F12, F10, F2
      ADDD   F16, F14, F2
      SD     0(R1), F4
      SD     -8(R1), F8
      SUBI   R1, R1, #32
      SD     16(R1), F12
      BEQZ   R1, loop
      SD     8(R1), F16
```

Dependences and Parallelism

Data Dependence (flow or RAW dependence)

- Instruction i is data dependent on instruction j if it uses the result produced by instruction j.
- It is a true dependence indicating the data flow between the two instructions and cannot be eliminated.
  - It is a property of programs which limits the amount of parallelism we can explore.
  - Pipeline organization determines whether they cause RAW hazards or stalls.
- It occurs either through
  - a register, or
  - memory location
- Detecting data dependences with registers is easier than those with memory locations.

Name dependences

- Anti-dependence (WAR dependence)
  - Instruction i is anti-dependent on instruction j if it writes a register or memory location after instruction j reads the same register or memory location.
- Output (WAW) dependence
  - Instruction i is output dependent on instruction j if it writes the register or memory location after instruction j writes the same register or memory location.
- Name dependences are caused by using the same register or memory location.
- They can be eliminated by using a different register or memory location for instruction i to write.

Control Dependence

- Instruction i is control dependent on instruction j if whether it is executed depends on the outcome of instruction j.
- Instruction j is typically a branch instruction.
- Control dependence is preserved by
  - executing instructions in order and
  - detecting the control hazard and delaying the branch target instruction.

- Control dependence preserves
  - exception behavior

```
        BEQZ    R2, L1
        LW      R1, 0(R2)
    L1: ...
```

    The exception with `LW` should never happen.

  - data dependence

```
        ADD  R1, R2, R3
        BEQZ R4, L
        SUB  R1, R5, R6
    L:  OR   R7, R1, R8
```

    `OR` data depends on `ADD` or `SUB`, if `BEQZ` is taken or untaken, respectively.

Dynamic Scheduling

What we have done with dependences so far

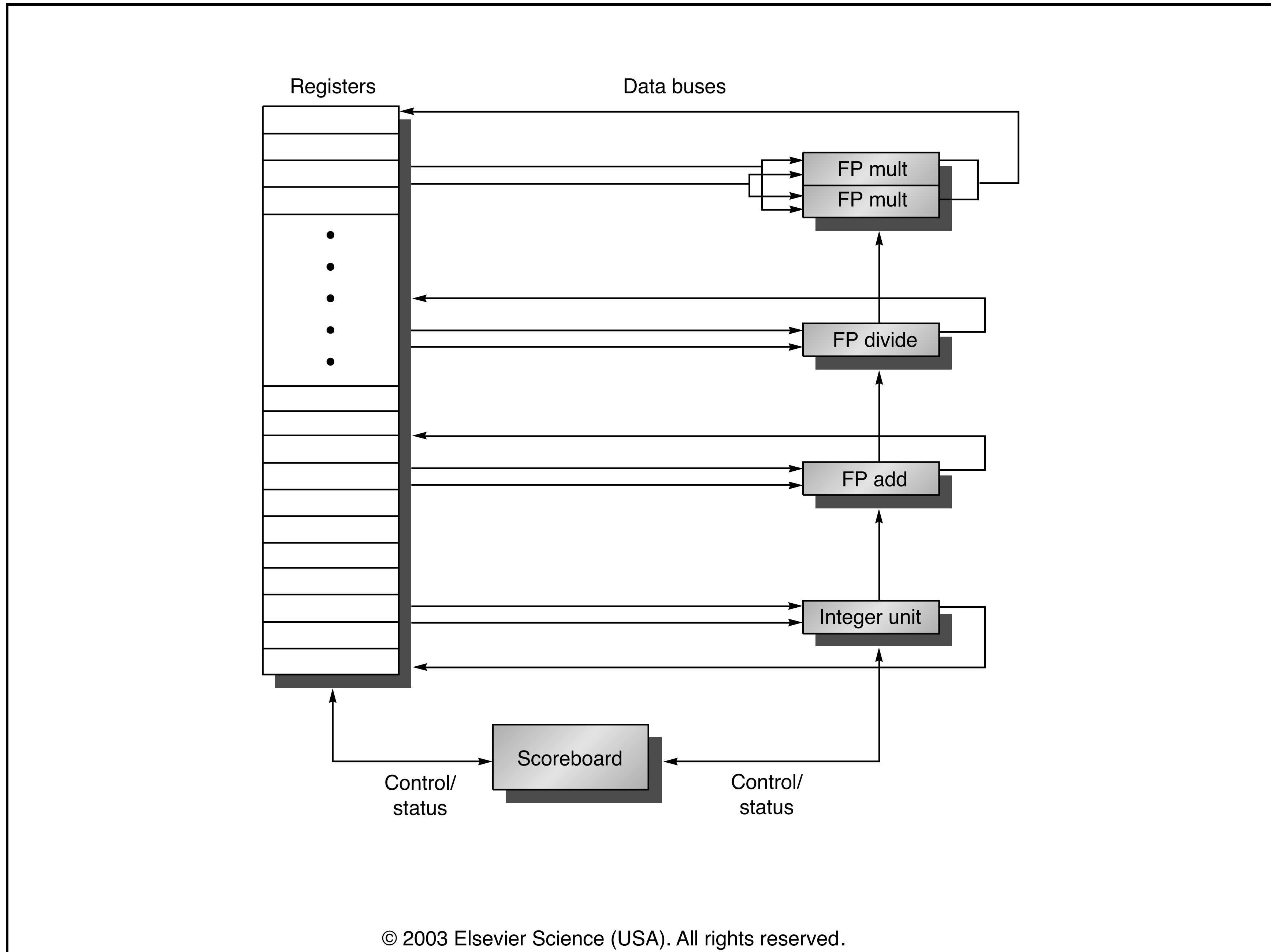
- Basic MIPS pipeline with in-order issue
  - Instructions are issued in the order of the code sequence.
  - If an instruction is stalled due to dependences, all the subsequent instructions are also stalled.
- Static pipeline scheduling reduces data stalls by re-ordering the instructions at compile-time. However,
  - it limits the binary compatibility – old binary code cannot run on new pipelines efficiently;
  - it is difficult for compilers to analyze and detect dependences with memory locations in the instruction level.

Dynamic scheduling

- detects and enforces dependences at run-time in hardware including those with memory locations,
- does out-of-our execution of instructions,
- decouples instruction issue and operands reading, and
- complicates hardware implementation of the pipeline
- Two schemes
  - Scoreboard originated from CDC 66000
  - Tomasulo approach in IBM 360/91

Scoreboard

- Detects and enforces all dependences including name dependences
- Decouples the instruction issue and the operand read from registers
- Centralized control through the scoreboard
- Multiple data channels between the register file and the functional units
- Basic structure of MIPS processor with a scoreboard (Figure A.51)



© 2003 Elsevier Science (USA). All rights reserved.

Four steps of scoreboard to replace ID, EX and WB

- Issue — delay issuing the instruction if there is a
  - structural hazard or
  - WAW hazard
- Read operands — delay reading the operands until they are available in the registers if there is a
  - RAW hazard
- Execution — no delay once started, update the scoreboard at completion.
- Write result — delay writing the register until the WAR hazard is cleared, if any.

Data structure of scoreboard

- Instruction Status – record which step the instruction is in
- Functional unit status – for each functional unit
  - Busy: busy or not
  - Op : the operation to perform
  - F_i: destination register
  - F_j, F_k: source registers
  - Q_j, Q_k: functional units producing F_j, F_k
  - R_j, R_k: flags F_j, F_k
    - FALSE(0), if F_j, F_k are targets of a functional unit, but the result is not ready yet;
    - TRUE(non-zero), if F_j, F_k are not targets of any functional unit, or data in F_j, F_k is written and ready.
- Register result status – for each register. It contains
  - the functional unit name (non-zero) which will write the register;
  - zero if no functional unit will write the register;
  - Used as both name and flag

Scoreboard Algorithm

- Assume
  - `FU`: the function unit of the instruction
  - `D`: the name of the destination register
  - `S1` and `S2`: the names of the source registers

Instruction status	Wait until	Bookkeeping
Issue	not Busy[FU] $\wedge$ not Result[D]	Busy[FU] $\leftarrow$ true; Op[FU] $\leftarrow$ op; Fi[FU] $\leftarrow$ D; Fj[FU] $\leftarrow$ S1; Fk[FU] $\leftarrow$ S2; Qj[FU] $\leftarrow$ Result[S1]; Qk[FU] $\leftarrow$ Result[S2]; Rj[FU] $\leftarrow$ not Qj[FU]; Rk[FU] $\leftarrow$ not Qk[FU]; Result[D] $\leftarrow$ FU;
Read operands	Rj[FU] $\wedge$ Rk[FU]	Rj[FU] $\leftarrow$ false; Rk[FU] $\leftarrow$ false; Qj $\leftarrow$ 0; Qk $\leftarrow$ 0;
Execution	Functional unit done	
Write result	$\forall$f((Fj[f] $\neq$Fi[FU] $\vee$ Rj[f]==false) $\wedge$ (Fk[f] $\neq$Fi[FU] $\vee$ Rk[f]==false))	$\forall$f(if Qj[f]==FU then Rj[f] $\leftarrow$true) $\forall$f(if Qk[f]==FU then Rk[f] $\leftarrow$true) Result[Fi[FU]] $\leftarrow$0; Busy[FU] $\leftarrow$false;

Achievements of Scoreboard

- Instructions can be issued as long as there are no structural hazards and WAW hazards.
- Independent instructions can be executed out-of-order.
- Improves the performance without compiler static pipeline scheduling.