Lecture 2

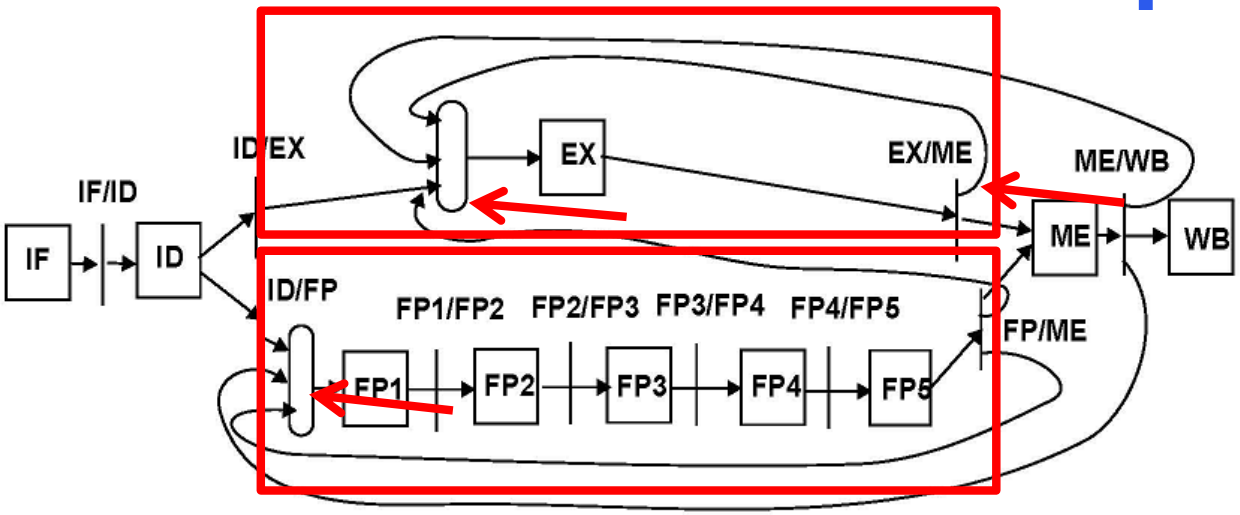
Instruction scheduling techniques

→ **Statically scheduled pipelines (Ch 3.3.2 - 3.3.6)**

- ✓ Out-of-order instruction completion (Ch 3.3.2)
- ✓ Superpipelined and superscalar CPUs (Ch 3.3.3)
- ✓ Static instruction scheduling (Ch 3.3.4-3.3.5)

Pipelines with Out-of-Order Completion (Ch 3.3.2)

Out-of-order Instruction Completion



- Floating-point (FP) instructions take 5 clocks and are pipelined
- Integer unit: Handles integer inst., branches and Loads/Stores
- Two separate register files: integer and FP
- Forwarding paths into EX and FP1
- Instructions wait in ID until they can proceed without data hazards (RAW and WAW)
- Still **in-order execution** but **out-of-order completion**

Latency vs Repeat Interval

- **Latency of operation:**
  - **Minimum number of cycles between an instruction producing a result and the instruction consuming it**
  - **Dependent instruction must stall in ID until its operand is forwarded**
  - **If functional units are linear pipelines then operation latency is execution time minus 1**
- **Repeat/Inititation interval:**
  - **Number of cycles that must elapse between consecutive issues of instructions to the same execution unit**
  - **If a functional unit is not pipelined, two consecutive instructions may not be issued to it in consecutive cycles because of structural hazards**

For the new FP capable pipeline:

FUNCTIONAL UNITS	LATENCY	INITIATION INTERVAL
INTEGER ALU	0	1
LOAD	1	1
FP OP	4	1(5 IF NOT PIPELINED)

New Structural Hazards

	CLOCK ==>	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
I1	ADD.S F1,F2,F1	IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB	
I2	ADD.S F4,F2,F3		IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB
I3	L.S F10			IF	ID	EX	ME	WB			
I4	L.S F12				IF	ID	EX	ME	WB		
I5	L.S F14					IF	ID	EX	ME	WB	

Approach: scan columns for common resource usage:

- At cycle C8: I1 and I5 are both in the ME stage (but I1 doesn't access memory)
- At cycle C9: I1 and I5 are both in the WB stage => structural hazard!

Avoiding structural hazard in the case of WB

- Add resources: another write port to the FP register file
- Stall one of the instructions (i.e., I5) in ID

Also: Structural hazards on execution units

Two general techniques to avoid structural hazards

- Add hardware resources
- Deal with it by stalling instructions to serialize conflicts

New Data Hazards

WAW hazards on FP registers are possible since instructions now reach WB out of (process/program) order

```
ADD.D  F2,F4,F6
L.D    F2,0(R2)
```

- **The LD must stall in ID**

NOTE: WAR hazards are not an issue because reads happen early

More RAW hazards due to longer operation latency:

	CLOCK ==>	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
I1	L.D F4, 0(R2)	IF	ID	EX	ME	WB						
I2	MULT.D F0,F4,F6		IF	ID	ID	FP1	FP2	FP3	FP4	FP5	ME	WB
I3	S.D F0, 0(R2)			IF	IF	ID	ID	ID	ID	ID	EX	ME

No data hazards on memory instructions

Loads/Stores follow the same pipeline path

Control hazards are unchanged

Quiz 2.1

	CLOCK->	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
I1	MULT.D F1,F2,F3	IF	ID	FP1	FP2	FP3	FP4	ME	WB			
I2	MULT.D F4,F1,F2		IF	ID	ID	ID	ID	FP1	FP2	FP 3	FP 4	ME
I3	ADD R1,R2,R3			IF	ID	EX	ME	WB				
I4	ADD R4,R1, R6				IF	ID	EX	ME	WB			

Which of the following statements are correct

a) There is potentially a structural hazard in C8 for the registerfile

b) Forwarding is needed between I2 and I3

c) Forwarding is needed between I1 and I2

d) Forwarding is needed between I3 and I4

Coping with Precise Exceptions

Precise Exceptions

Consider the following code:

I1: ADD.S F1, F2, F1 ***floating-point ADD

I2: ADD R2,R1,R3 ***integer ADD

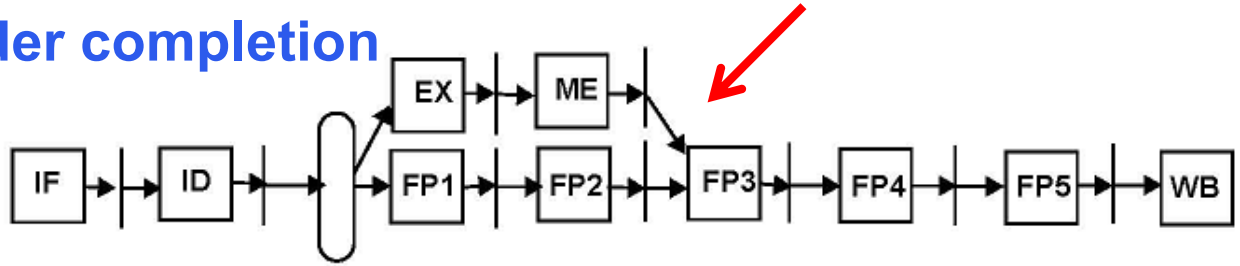
Exceptions may happen out of process/program order (I2 before I1)!

	C1	C2	C3	C4	C5	C6	C7	C8	9	10	11
ADD.S F1,F2,F1	IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB		
ADD R2,R1,R3		IF	ID	EX	ME	WB					

- I2 stores its result in C6
- I1 causes an exception in C6
- **Problem:** R2 has already been modified!

Coping with Precise Exceptions

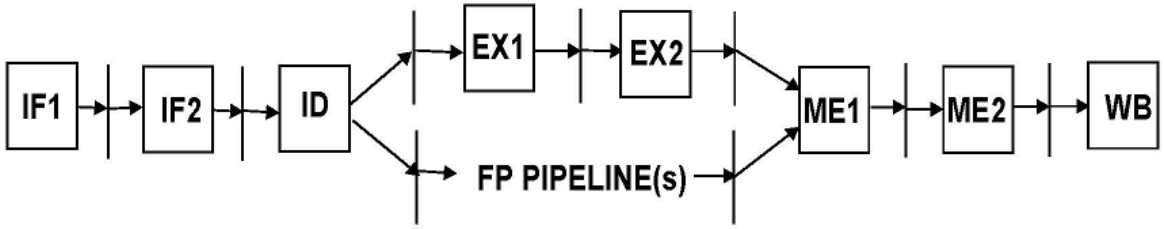
- **Forget precision on exceptions**
  - Not viable today in machines with virtual memory and IEEE FP standard
  - Give state to software and let software figure it out. Not feasible for complex pipelines.
  - Support two operational modes:
    - SLOW: No pipelining, all exceptions enabled (debugging mode)
    - FAST: Pipelining, some exceptions are disabled.
- **Deal with exceptions as if they were hazards**
  - Do not issue an instruction in ID until it is sure that all prior instructions are exception free
  - Detect exceptions as early as possible in the execution
  - May stiffle pipelining
- **Force in-order completion**



Superpipelining and Superscalar (Ch 3.3.3)

CHALMERS
Chalmers University of Technology

Superpipelined CPU

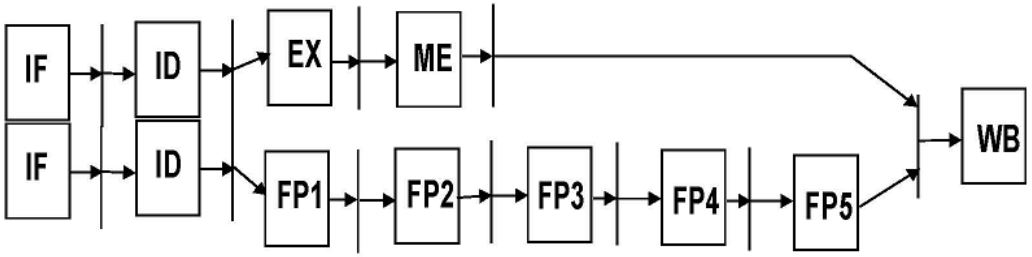


- Some stages in the 5-stage pipeline are further pipelined
  - to increase clock rate
  - Here, IF, EX and ME are now 2 pipeline stages
  - Clock is 2X as faster
- BUT: There is no ”free lunch” though:
  - Branch penalty when taken is now 3 clocks
  - Latencies counted in clock cycles are higher

Assuming unchanged 5-stages for FP

FUNCTIONAL UNITS	LATENCY	INITIATION INTERVAL
INTEGER ALU	1	1
LOAD	3	1
FP OP	4	1(5 IF NOT PIPELINED)

Superscalar CPU



- Fetch, decode and execute up to 2 instructions per clock
  - Same clock rate as basic pipeline
  - Easy (in this case) because of the 2 (INT and FP) register files
- Issue a pair of instructions:
  - Pair must be integer/branch/ memory AND FP (compiler can do that)
  - Instruction pair must be independent and have no hazard with prior instr.
  - Same latencies as basic pipeline; branches execute in EX
  - Exceptions are more complex to deal with
- Hard to build more than 2-way
- IPC (instructions per clock) instead of CPI (IPC=1/CPI)

Add superpipelining to superscalar

Quiz 2.2

	CLOCK->	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
I1	DIV.D F1,F2,F3	IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB		
I2	ADD R1,R2,R3	IF	ID	EX	ME	WB						
I3	MULT.D F4,F5,F6		IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB	
I4	ADD R4,R1, R6		IF	ID	EX	ME	WB					

Which of the following statements are correct

a) Instructions finish in this order: I1, I2, I3 and I4

b) Instructions finish in this order: I2, I4, I1, I3

c) If the registerfile has two write ports I1 and I3 both finish in C9

d) A single-issue pipeline would finish the sequence in C11

Static Instruction Scheduling (Ch 3.3.4)

CHALMERS
Chalmers University of Technology

Static Branch Prediction

Hardwired branch prediction

- Always predict untaken and execute in EX
- Branches at the bottom of a loop are mostly mispredicted
- No compiler assist is possible
- Could predict taken, but not very useful here (Why?)
- Decisions made at design time (benchmarking)

Compile-time branch prediction

- Each branch instr. has a ”hint” bit set by the compiler
- Compiler profiles the code and sets the hint bit
- Taken/Not taken prediction is much more flexible

Static Instruction Scheduling

To maximize instruction throughput (IPC) of static pipelines

Compiler schedules instructions in code chunks

- **Local (basic block level)**
- **Global (across basic blocks):**
  - **Cyclic (loops)**
  - **Non-cyclic (traces)**

Local Scheduling Example 1(2)

```
for (i=0;i<100;i++)
    A[i]:= A[i] + B[i];
```

An unoptimized code for the loop in the basic pipeline:

```
Loop    L.S  F0,0(R1)     (1)
        L.S  F1,0(R2)     (1)
        ADD.S F2,F1,F0    (2)
        S.S  F2,0(R1)     (5)
        ADDI R1,R1,#4     (1)
        ADDI R2,R2,#4     (1)
        SUBI R3,R3,#1     (1)
        BNEZ R3,Loop      (3)
```

Two pairs of instructions with RAW dependencies between them

Total execution time : 15 clocks; CPI = 15/8 = 1.88

Local Scheduling Example 2(2)

Compiler moves instructions inside the loop body:

- **Move SUBI up**
- **Move S.S down**

```
Loop L.S  F0,0(R1)      (1)
     L.S  F1,0(R2)      (1)
     SUBI R3,R3,1       (1)
     ADD.S F2,F1,F0     (1)
     ADDI R1,R1,#4      (1)
     ADDI R2,R2,#4      (1)
     S.S  F2,-4(R1)     (3)
     BNEZ R3,Loop       (3)
```

- **Must change the displacement of the Store (WAR in R1)**
- **Execution time is 12 clocks, a speed up of 15/12 = 1.25**

Local scheduling scope is too limited

Quiz 2.3

```
I1: L.S F0,0(R1)     (1)
I2: L.S F1,0(R2)     (1)
I3: ADD.S F2,F1,F0   (2)
I4: S.S F2,0(R1)     (5)
I5: ADDI R1,R1,#4    (1)
I6: ADDI R2,R2,#4    (1)
I7: SUBI R3,R3,#1    (1)
I8: BNEZ R3,Loop     (3)
```

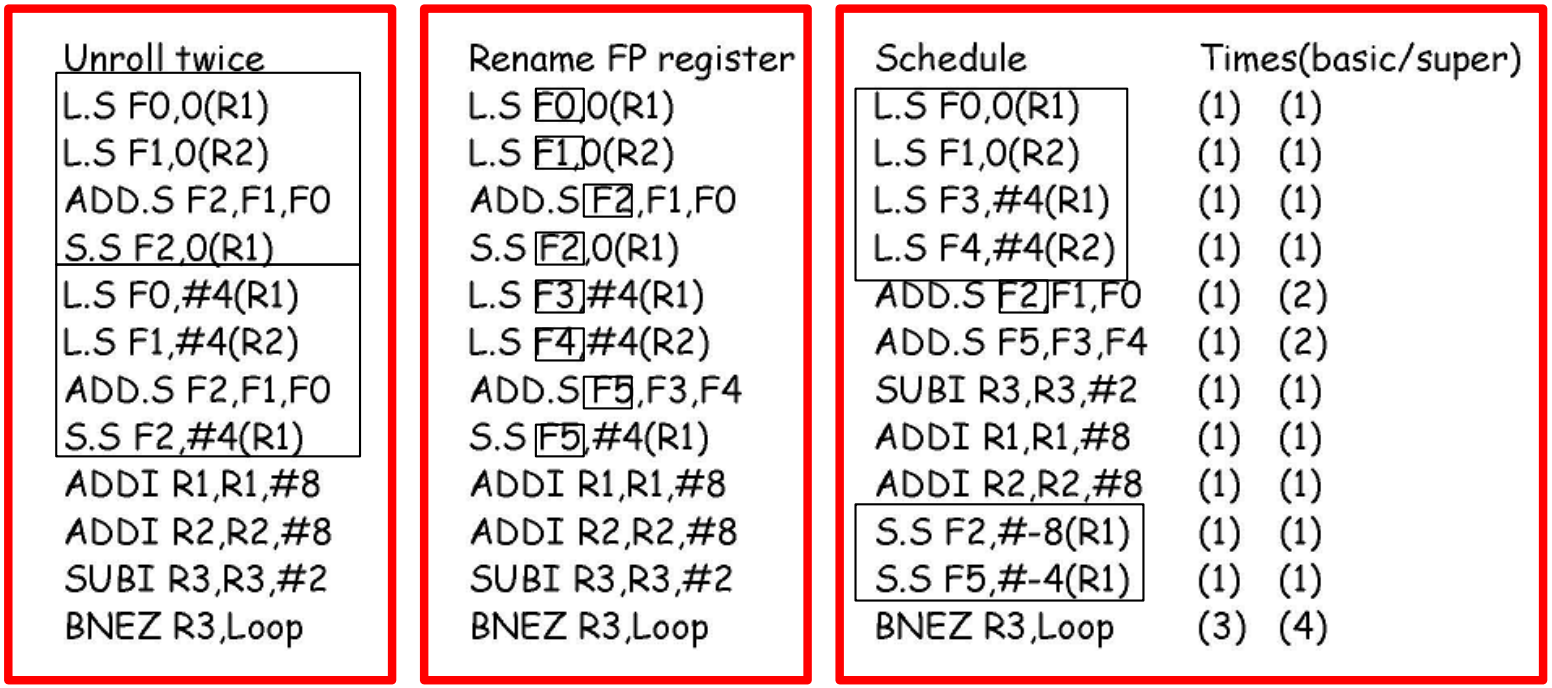
Which of the following transformations are correct

a) Moving I7 before I1
b) Moving I6 before I2
c) Moving I6 before I3
d) Moving I4 after I5 without changing it
e) Moving I4 after I5 and changing it to S.S F2,#-4(R1)

Global Static Instruction Scheduling: Loop Unrolling (Ch 3.3.5)

CHALMERS
Chalmers University of Technology

Loop Unrolling 1(2)



- **Total time (Basic): 14 clocks or 7 clocks per iteration of original**
- **Loop speedup: 15/7 = 2.14**

Superpipelined processor (2nd column)

- **Total time: 17 clocks or 8.5 clocks per iteration of orig. loop**
- **Clock rate is 2X as fast. Speedup: 2X 15/8.5 = 3.52**

Loop Unrolling 2(2)

Superscalar processor

Schedule		Time	Program
L.S F0,0(R1)		(1)	L.S F0,0(R1)
L.S F1,0(R2)		(1)	L.S F1,0(R2)
L.S F3,#4(R1)		(1)	L.S F3,#4(R1)
L.S F4,#4(R2)		(1)	L.S F4,#4(R2)
SUBI R3,R3,#2	ADD.S F2,F1,F0	(1)	SUBI R3,R3,#2
ADDI R1,R1,#8	ADD.S F5,F3,F4	(1)	ADD.S F2,F1,F0
ADDI R2,R2,#8		(1)	ADDI R1,R1,#8
S.S F2,#-8(R1)		(3)	ADD.S F5,F3,F4
S.S F5,#-4(R1)		(1)	ADDI R2,R2,#8
BNEZ R3,Loop		(3)	S.S F2,#-8(R1)
			S.S F5,#-4(R1)
			BNEZ R3,Loop

Limited opportunities because of the unbalance between integer and FP instructions

Limitations of Loop Unrolling

Works well if loop iterations are independent

```
for (i=5;i<100;i++)
    A[i]:= A[i-5] + B[i];
```

Loop-carried RAW dependency recurrence with distance 5 iter.

- **Unroll loop 4 times**
- **5 times => Loop carried dependency limits code motion**

Consumes architectural registers for *renaming*

Code expansion

- **Affects instruction cache (I-cache) and memory**

Problem when the number of iterations is unknown at compile time.

Global Static Instruction Scheduling: Software Pipelining (Ch 3.3.5)

CHALMERS
Chalmers University of Technology

Software Pipelining

Original loop is translated into a pipelined loop

- **Pipelines dependent instructions in different iterations**

Application to the simple loop:

- **Since the ADD.S is the long latency operation, we can pipeline the two Loads and the ADD.S with the Store**

	O_ITE1	O_ITE2	O_ITE3	O_IT4
Prologue	L.S F0,0(R1) L.S F1,0(R2) ADD.S F2,F1,F0	---	---	---
P_ITE1	S.S F2,0(R1)	L.S F0,0(R1) L.S F1,0(R2) ADD.S F2,F1,F0	---	---
P_ITE2	---	S.S F2,0(R1)	L.S F0,0(R1) L.S F1,0(R2) ADD.S F2,F1,F0	---
P_ITE3	---	---	S.S F2,0(R1)	L.S F0,0(R1) L.S F1,0(R2) ADD.S F2,F1,F0
Epilogue				S.S F2,0(R1)

- NOTE THAT EXACTLY THE SAME OPERATIONS ARE EXECUTED IN EXACTLY THE SAME ORDER IN BOTH THE ORIGINAL LOOP AND THE PIPELINED LOOP
- THE CODE IS SIMPLY REORGANIZED

Software Pipelining Example

```
Prologue: L.S  F0,0(R1)
          L.S  F1,0(R2)
          SUBI R3,R3,1
          ADD.S F2,F1,F0
          ADDI R1,R1,#4
          ADDI R2,R2,#4
Loop      S.S  F2,#-4(R1)  (1)
          L.S  F0,0(R1)    (1)
          L.S  F1,0(R2)    (1)
          SUBI R3,R3,1     (1)
          ADD.S F2,F1,F0   (1)
          ADDI R1,R1,#4    (1)
          ADDI R2,R2,#4    (1)
          BNEZ R3,Loop     (3)
Epilogue: S.S  F2,#-4(R1)
```

- No stalls (base), no code expansion; no register renaming
- Use loop unrolling and then software pipelining

Limits of Static Pipelines

Strengths

- Hardware simplicity: clock rate advantage over complex designs
- Predictable: static performance predictions are reliable
- Compiler has global view, e.g., optimize loops
- Power/energy advantage
- Good target for embedded systems

Weaknesses

- Dynamic events (cache misses and cond. branches)
- Freeze the processor on a miss (can't deal with latency tolerance)
- Lack of dynamic information (memory addresses)
- No good solution for precise exceptions

We consider dynamically scheduled pipelines next

Quiz 2.4

```
LOOP: L.S F0,0(R1)
      L.S F1,0(R1)
      ADD.S F2,F1,F0
      S.S F2,0(R1)
      ADDI R1,R1,#4
      SUBI R3,R3,#1
      BNEZ R3,LOOP
```

```
LOOP: L.S F0,0(R1)
      L.S F1,0(R1)
      L.S F2,#4(R1)
      L.S F3 #4(R1)
      ADD.S F2,F1,F0
      ADD.S F4,F2,F3
      ADDI R1,R1,#4
      SUBI R3,R3,#2
      S.S F2,0(R1)
      S.S F4,0(R1)
      BNEZ R3,LOOP
```

What is wrong with the unrolled loop

a) L.S F2,#4(R1) should be L.S F2,0(R1)

b) ADDI R1,R1,#4 should be ADDI R1,R1,#8

c) S.S F2,0(R1) should be S.S F2,#-8(R1)

d) S.S F4,0(R1) should be S.S F4,#-4(R1)

What you should know by now

Statically scheduled pipelines

- How out-of-order completion can lower CPI
- Static instruction scheduling techniques
  - Local: Reorder instructions inside a basic block
  - Loop unrolling
  - Software pipelining
- Superpipelined and superscalar static pipelines
- Exception handling in static pipelines