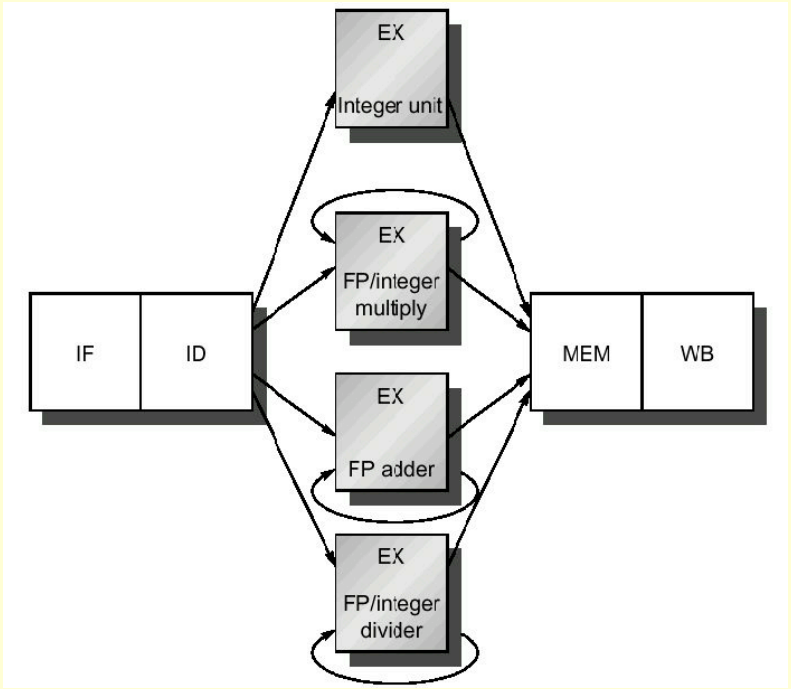
CMSC 611: Advanced Computer Architecture

Instruction Level Parallelism

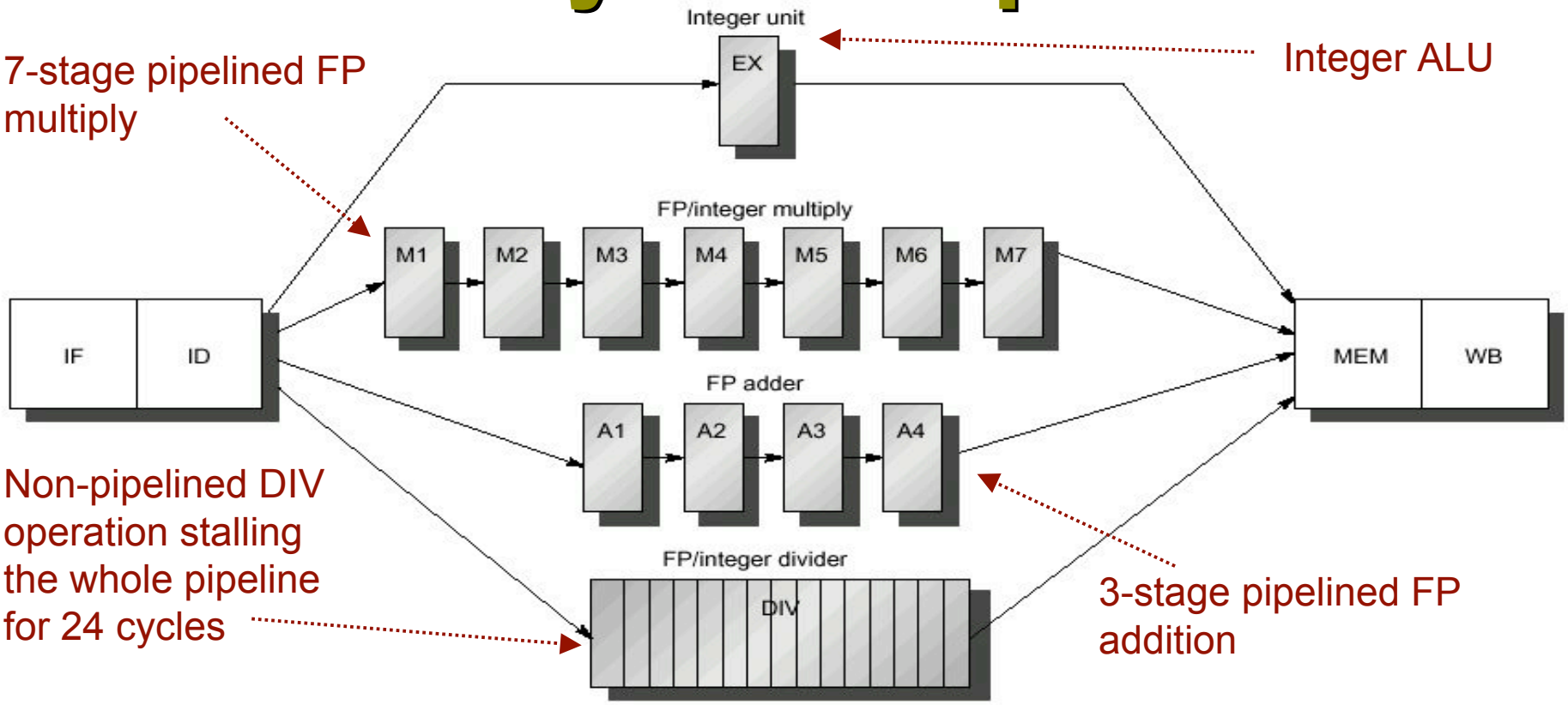
Some material adapted from Mohamed Younis, UMBC CMSC 611 Spr 2003 course slides

Floating-Point Pipeline

- Impractical for FP ops to complete in one clock
  - (complex logic and/or very long clock cycle)
- More complex hazards
  - Structural
  - Data



Multi-cycle FP Pipeline



MULTD	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			

Example: *blue indicate where data is needed and red when result is available*

Multi-cycle FP: EX Phase

- Latency: cycles between instruction that produces result and instruction that uses it
  - Since most operations consume their operands at the beginning of the EX stage, latency is usually number of the stages of the EX an instruction uses
- Long latency increases the frequency of RAW hazards
- Initiation (Repeat) interval: cycles between issuing two operations of a given type

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

FP Pipeline Challenges

- Non-pipelined divide causes structural hazards
- Number of register writes required in a cycle can be larger than 1
- WAW hazards are possible
  - Instructions no longer reach WB in order
- WAR hazards are **NOT** possible
  - Register reads are still taking place during the ID stage
- Instructions can complete out of order
  - Complicates exceptions
- Longer latency makes RAW stalls more frequent

Instruction	Clock cycle number																
	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												
MULTD 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

Example of RAW hazard caused by the long latency

Structural Hazard

Instruction	Clock cycle number										
	1	2	3	4	5	6	7	8	9	10	11
MULTD F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
…		IF	ID	EX	MEM	WB					
…			IF	ID	EX	MEM	WB				
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
…					IF	ID	EX	MEM	WB		
…						IF	ID	EX	MEM	WB	
LD F2, 0(R2)							IF	ID	EX	MEM	WB

- At cycle 10, MULTD, ADDD and LD instructions all in MEM
- At cycle 11, MULTD, ADDD and LD instructions all in WB
  - Additional write ports are not cost effective since they are rarely used
- Instead
  - Detect at ID and stall
  - Detect at MEM or WB and stall

WAW Data Hazards

Instruction	Clock cycle number										
	1	2	3	4	5	6	7	8	9	10	11
MULTD F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
…		IF	ID	EX	MEM	WB					
…			IF	ID	EX	MEM	WB				
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
…					IF	ID	EX	MEM	WB		
LD F2, 0(R2)						IF	ID	EX	MEM	WB	
….							IF	ID	EX	MEM	WB

- WAW hazards can be corrected by either:
  - Stalling the latter instruction at MEM until it is safe
  - Preventing the first instruction from overwriting the register
- Correcting at cycle 11 OK unless intervening RAW/use of F2
- WAW hazards can be detected at the ID stage
  - Convert 1st instruction to no-op
- WAW hazards are generally very rare, designers usually go with the simplest solution

Detecting Hazards

- Hazards among FP instructions & and combined FP and integer instructions
- Separate int & fp register files limits latter to FP load and store instructions
- Assuming all checks are to be performed in the ID phase:
  - Check for structural hazards:
    - Wait if the functional unit is busy (Divides in our case)
    - Make sure the register write port is available when needed
  - Check for a RAW data hazard
    - Requires knowledge of latency and initiation interval to decide when to forward and when to stall
  - Check for a WAW data hazard
    - Write completion has to be estimated at the ID stage to check with other instructions in the pipeline
- Data hazard detection and forwarding logic from values stored between the stages

Maintaining Precise Exceptions

- Pipelining FP instructions can cause out-of-order completion
- Exceptions also a problem:

```
DIVF    F0, F2, F4
ADDF    F10, F10, F8
SUBF    F12, F12, F14
```

  - No data hazards
  - What if DIVF exception occurs after ADDF writes F10?

Four FP Exception Solutions

1. Settle for imprecise exceptions
   - Some supercomputers still uses this approach
   - IEEE floating point standard requires precise exceptions
   - Some machine offer slow precise and fast imprecise exceptions
2. Buffer the results of all operations until previous instructions complete
   - Complex and expensive design (many comparators and large MUX)
   - History or future register file

Four FP Exception Solutions

3. Allow imprecise exceptions and get the handler to clean up any miss
   - Save PC + state about the interrupting instruction and all out-of-order completed instructions
   - The trap handler will consider the state modification caused by finished instructions and prepare machine to resume correctly
   - Issues: consider the following example

     Instruction1: Long running, eventual exception

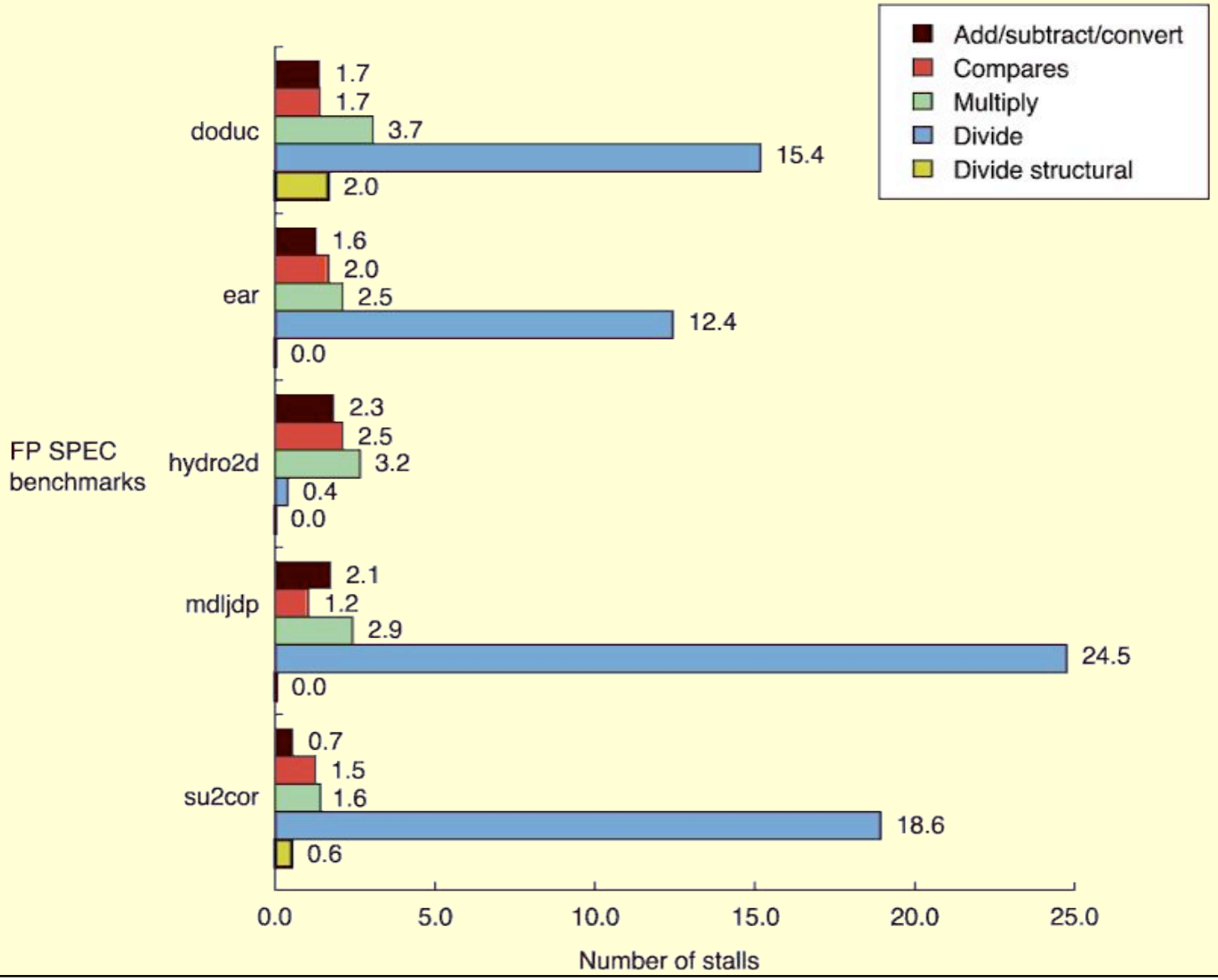
     Instructions 2 … (n-1) : Instructions that do not complete

     Instruction n : An instruction that is finished
   - The compiler can simplify the problem by grouping FP instructions so that the trap does not have to worry about unrelated instructions

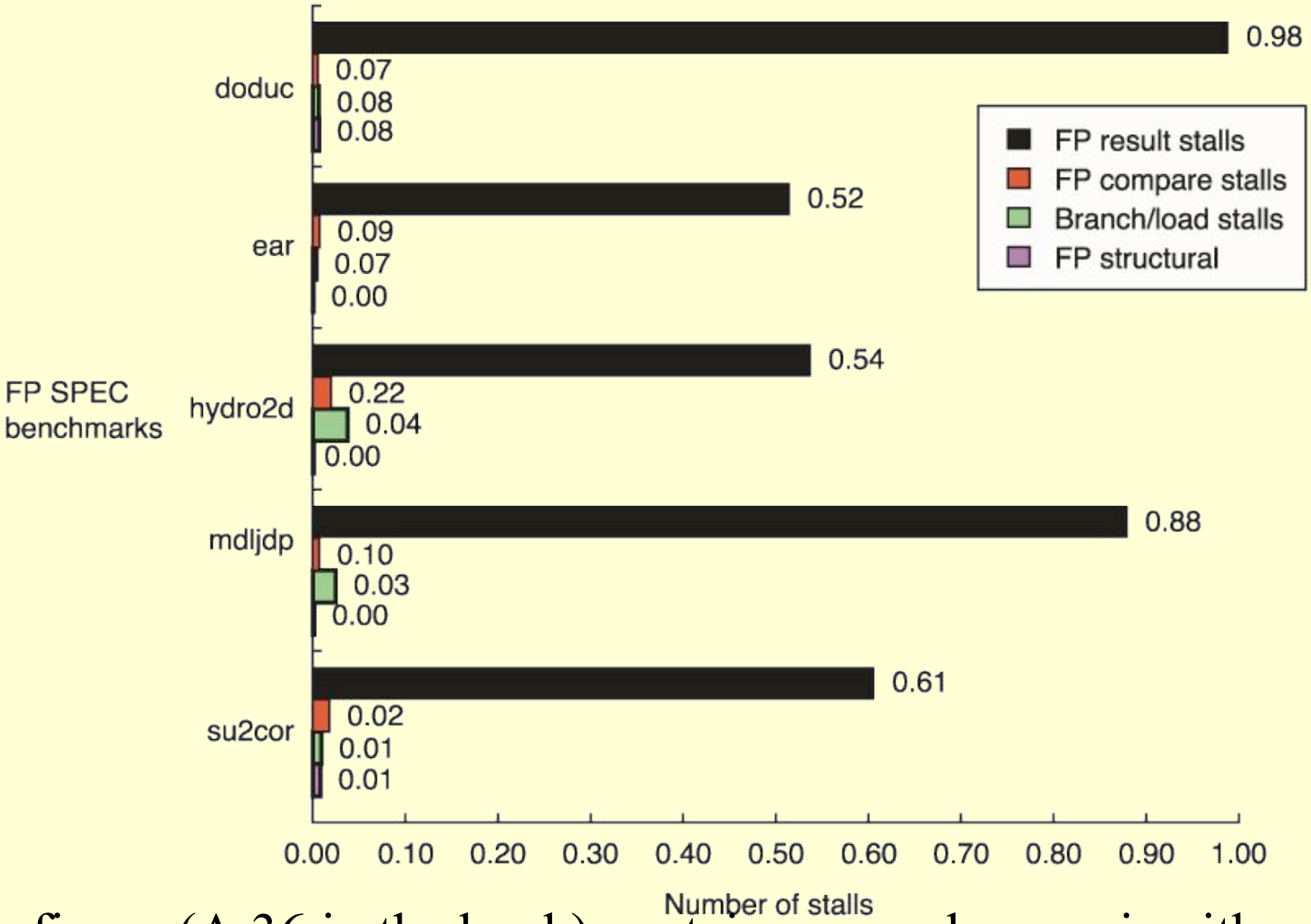
Four FP Exception Solutions

4. Allow instruction issue to continue only if previous instruction are guaranteed to cause no exceptions:
   - Mainly applied in the execution phase
   - Used on MIPS R4000 and Intel Pentium

Stalls/Instruction, FP Pipeline



More FP Pipeline Performance



This figure (A.36 in the book) contains several errors in either graph or data. Only take-home: result stalls are most common by far

Instruction Level Parallelism (ILP)

- Overlap the execution of unrelated instructions
- Both instruction pipelining and ILP enhance instruction throughput not the execution time of the individual instruction
- Potential of IPL within a basic block is very limited
  - in “gcc” 17% of instructions are control transfer meaning on average 5 instructions per branch

Loops: Simple & Common

```
for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + y[i];
```

- Techniques like loop unrolling convert loop-level parallelism into instruction-level parallelism
  - statically by the compiler
  - dynamically by hardware
- Loop-level parallelism can also be exploited using vector processing
- IPL feasibility is mainly hindered by data and control dependence among the basic blocks
- Level of parallelism is limited by instruction latencies

Major Assumptions

- Basic MIPS integer pipeline
- Branches with one delay cycle
- Functional units are fully pipelined or replicated (as many times as the pipeline depth)
  - An operation of any type can be issued on every clock cycle and there are no structural hazard

Instruction producing result	Instruction using results	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

Motivating Example

```
for (i=1000; i>0; i=i-1)
   x[i] = x[i] + s;
```

```
Loop:  LD     F0,x(R1)     ;F0=vector element
       ADDD   F4,F0,F2     ;add scalar from F2
       SD     x(R1),F4     ;store result
       SUBI   R1,R1,8      ;decrement pointer (DW)
       BNEZ   R1,Loop      ;branch R1!=zero
```

Standard Pipeline execution

```
Loop:  LD     F0,x(R1)
       stall
       ADDD   F4,F0,F2
       stall
       stall
       SD     x(R1),F4
       SUBI   R1,R1,8
       stall
       BNEZ   R1,Loop
       stall
```

Smart compiler

```
Loop:  LD     F0,x(R1)
       SUBI   R1,R1,8
       ADDD   F4,F0,F2
       stall  ;F4 latency
       BNEZ   R1,Loop
       SD     x+8(R1),F4
```

Sophisticated compiler optimization reduced execution time from 10 cycles to only 6 cycles

Loop Unrolling

```
Loop:  LD    F0,x(R1)
       ADDD  F4,F0,F2
       SD    x(R1),F4
       SUBI  R1,R1,8
       BNEZ  R1,Loop
```

Replicate loop body 4 times, will need cleanup phase if loop iteration is not a multiple of 4

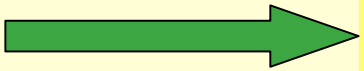
```
Loop:  LD    F0,x(R1)
       ADDD  F4,F0,F2
       SD    x(R1),F4    ;drop SUBI & BNEZ
       LD    F6,x-8(R1)
       ADDD  F8,F6,F2
       SD    x-8(R1),F8    ;drop again
       LD    F10,x-16(R1)
       ADDD  F12,F10,F2
       SD    x-16(R1),F12;drop again
       LD    F14,x-24(R1)
       ADDD  F16,F14,F2
       SD    x-24(R1),F16
       SUBI  R1,R1,#32   ;alter to 4*8
       BNEZ  R1,LOOP
```

- 6 cycles, but only 3 are loop body
- Loop unrolling limits overhead at the expense of a larger code
  - Eliminates branch delays
  - Enable effective scheduling
- Use of different registers needed to limit data hazard

Scheduling Unrolled Loops

Cycle		Instruction
1	Loop: LD	F0,x(R1)
3	ADDD	F4,F0,F2
6	SD	x(R1),F4
7	LD	F6,x-8(R1)
9	ADDD	F8,F6,F2
12	SD	x-8(R1),F8
13	LD	F10,x-16(R1)
15	ADDD	F12,F10,F2
18	SD	x-16(R1),F12
19	LD	F14,x-24(R1)
21	ADDD	F16,F14,F2
24	SD	x-24(R1),F16
25	SUBI	R1,R1,#32
27	BNEZ	R1,LOOP
28	stall	

Loop unrolling exposes more computation that can be scheduled to minimize the pipeline stalls



Understanding dependence among instructions is the key for for detecting and performing the transformation

Cycle		Instruction
1	Loop: LD	F0,x(R1)
2	LD	F6,x-8(R1)
3	LD	F10,x-16(R1)
4	LD	F14,x-24(R1)
5	ADDD	F4,F0,F2
6	ADDD	F8,F6,F2
7	ADDD	F12,F10,F2
8	ADDD	F16,F14,F
9	SD	x(R1),F4
10	SD	x-8(R1),F8
11	SUBI	R1,R1,#32
12	SD	x+16(R1),F12
13	BNEZ	R1,LOOP
14	SD	x+8(R1),F1

Inter-instruction Dependence

- Determining how one instruction depends on another is critical not only to the scheduling process but also to determining how much parallelism exists
- If two instructions are parallel they can execute simultaneously in the pipeline without causing stalls (assuming there is not structural hazard)
- Two instructions that are dependent are not parallel and their execution cannot be reordered

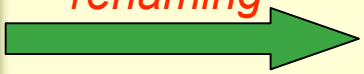
Dependence Classifications

- Data dependence (RAW)
  - Transitive: i → j → k = i → k
  - Easy to determine for registers, hard for memory
    - Does 100(R4) = 20(R6)?
    - From different loop iterations, does 20(R6) = 20(R6)?
- Name dependence (register/memory reuse)
  - Anti-dependence (WAR): Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first
  - Output dependence (WAW): Instructions i and j write the same register or memory location; instruction ordering must be preserved
- Control dependence, caused by conditional branching

Example: Name Dependence

```
Loop: LD    F0,x(R1)
      ADDD  F4,F0,F2
      SD    x(R1),F4
      LD    F0,x-8(R1)
      ADDD  F4,F0,F2
      SD    x-8(R1),F4
      LD    F0,x-16(R1)
      ADDD  F4,F0,F2
      SD    x-16(R1),F4
      LD    F0,x-24(R1)
      ADDD  F4,F0,F2
      SD    x-24(R1),F4
      SUBI  R1,R1,#32
      BNEZ  R1,Loop
```

Register renaming



```
Loop: LD    F0,x(R1)
      ADDD  F4,F0,F2
      SD    x(R1),F4
      LD    F6,x-8(R1)
      ADDD  F8,F6,F2
      SD    x-8(R1),F8
      LD    F10,x-16(R1)
      ADDD  F12,F10,F2
      SD    x-16(R1),F12
      LD    F14,x-24(R1)
      ADDD  F16,F14,F2
      SD    x-24(R1),F16
      SUBI  R1,R1,#32
      BNEZ  R1,LOOP
```

- Again Name Dependencies are Hard for Memory Accesses
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?
- Compiler needs to know that R1 does not change → *0(R1)≠ -8(R1)≠ -16(R1)≠ -24(R1)* and thus no dependencies between some loads and stores so they could be moved