CS425
Computer Systems Architecture

Fall 2021

Static Instruction Scheduling

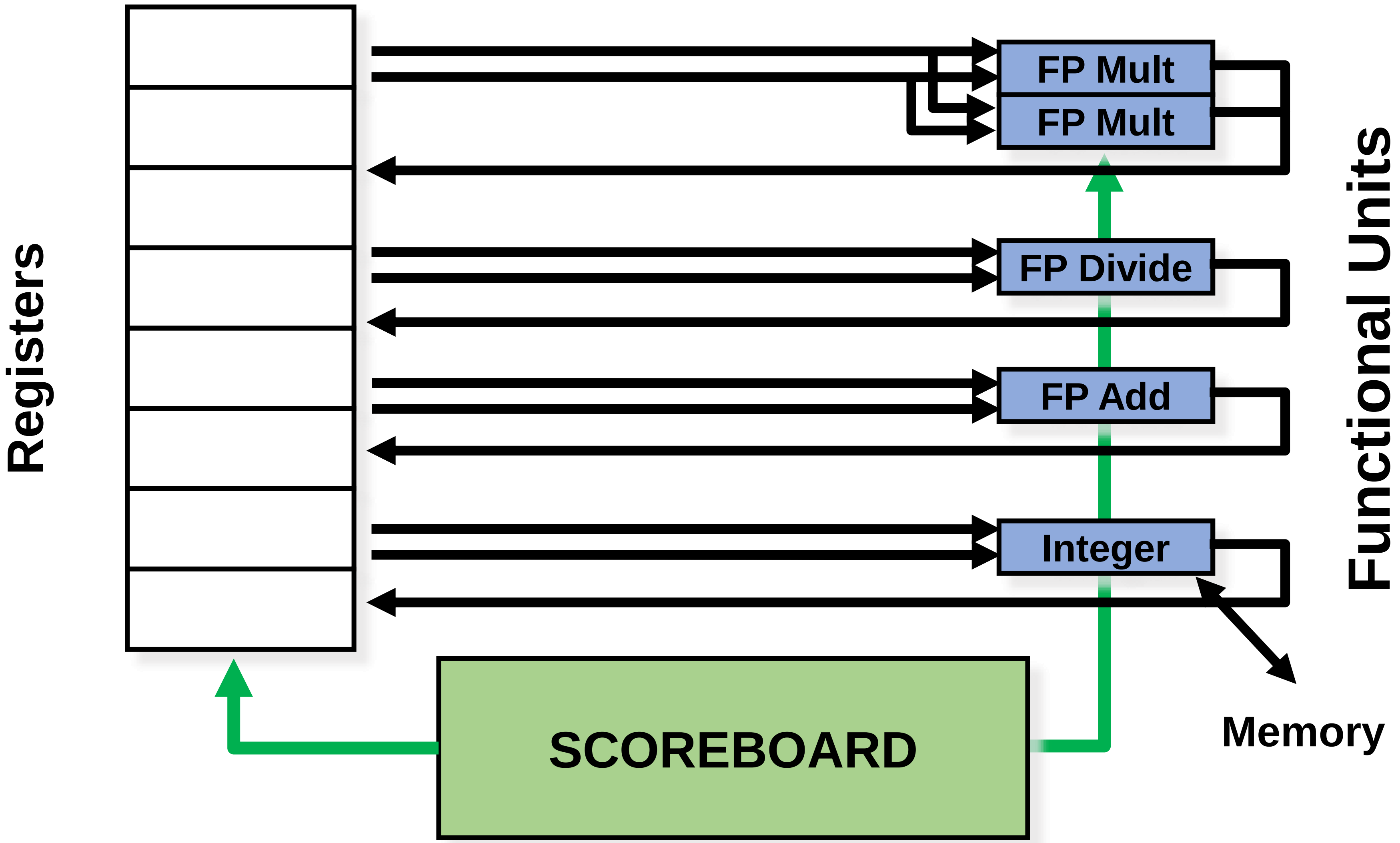
Techniques to reduce stalls

CPI = Ideal CPI + Structural stalls per instruction + RAW stalls per instruction + WAR stalls per instruction + WAW stalls per instruction

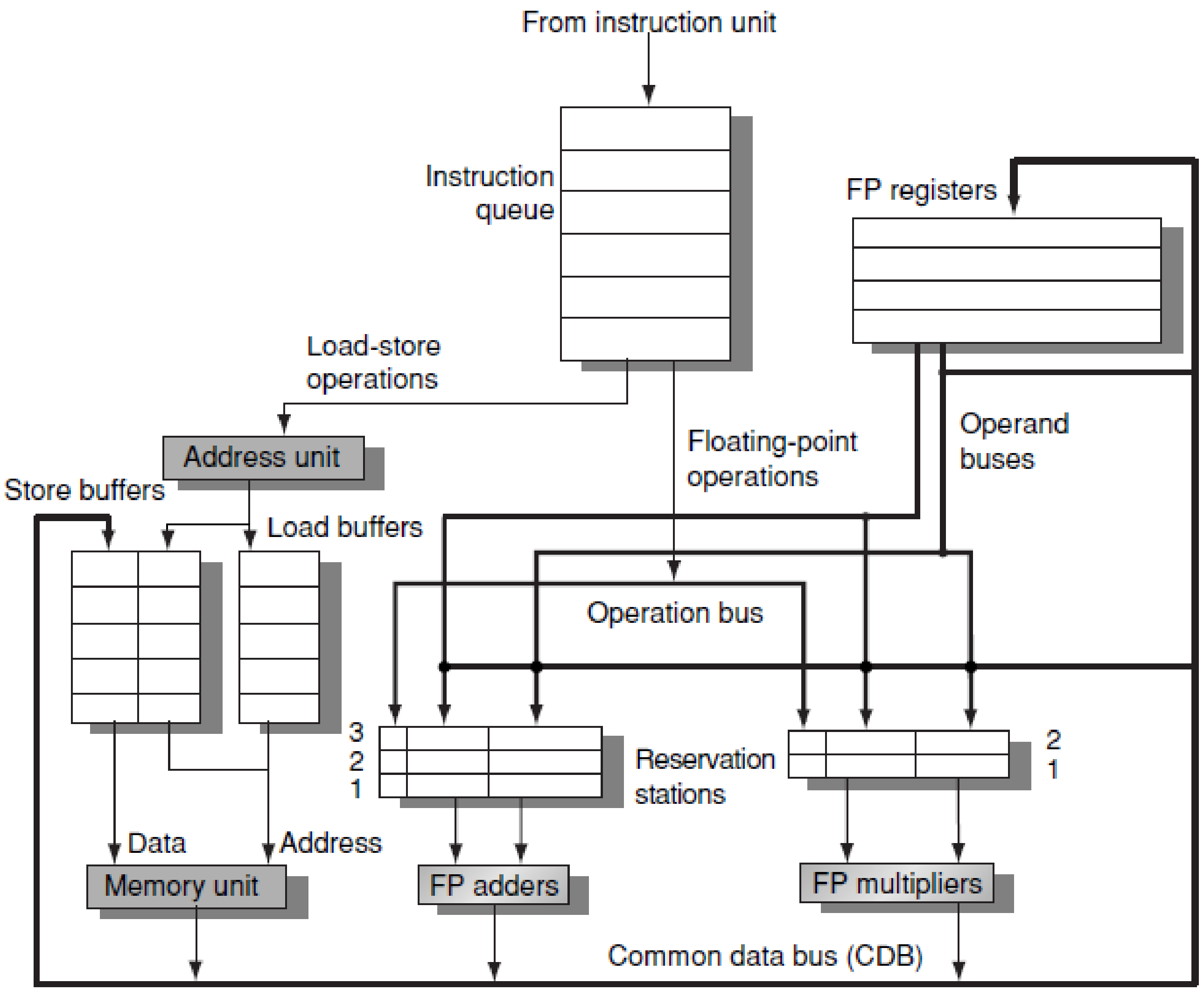
We will study two types of techniques:

Dynamic instruction scheduling	Static instruction scheduling (SW/compiler)
Scoreboard (reduce RAW stalls)	Loop Unrolling
Register Renaming (reduce WAR & WAW stalls) • Tomasulo • Reorder buffer	SW pipelining
Branch Prediction (reduce control stalls)	Trace Scheduling

Scoreboard Architecture (CDC 6600)



Tomasulo Organization



Dependencies between Instructions

- What are the sources of stalls/bubbles?
  - instructions that use the same registers
- ***Parallel*** instructions can execute without imposing any stalls (if we ignore structural hazards)
  - DIV.D F0, F2, F4
  - ADD.D F10, F1, F3
- ***Dependencies*** between instructions may lead to stalls
  - DIV.D F0, F2, F4

    RAW **must enter the execution stage in order**
  - ADD.D F10, F0, F3
- The dependencies between instructions limit the order of execution of these instructions (impose in order execution). In the 2nd example ADD.D **must** execute after DIV.D has completed. On the other hand, parallel instructions **may** execute in the any order (out-of-order execution). In the 1st example ADD.D can execute before DIV.D.

Dependencies between Instructions

- **Data Dependences** : instructions are data dependent when there is a chain of RAW hazards between them.

- **Name Dependences** : instructions are name dependent when there is a WAR (anti-dependence) or WAW (output-dependence) hazard between them.

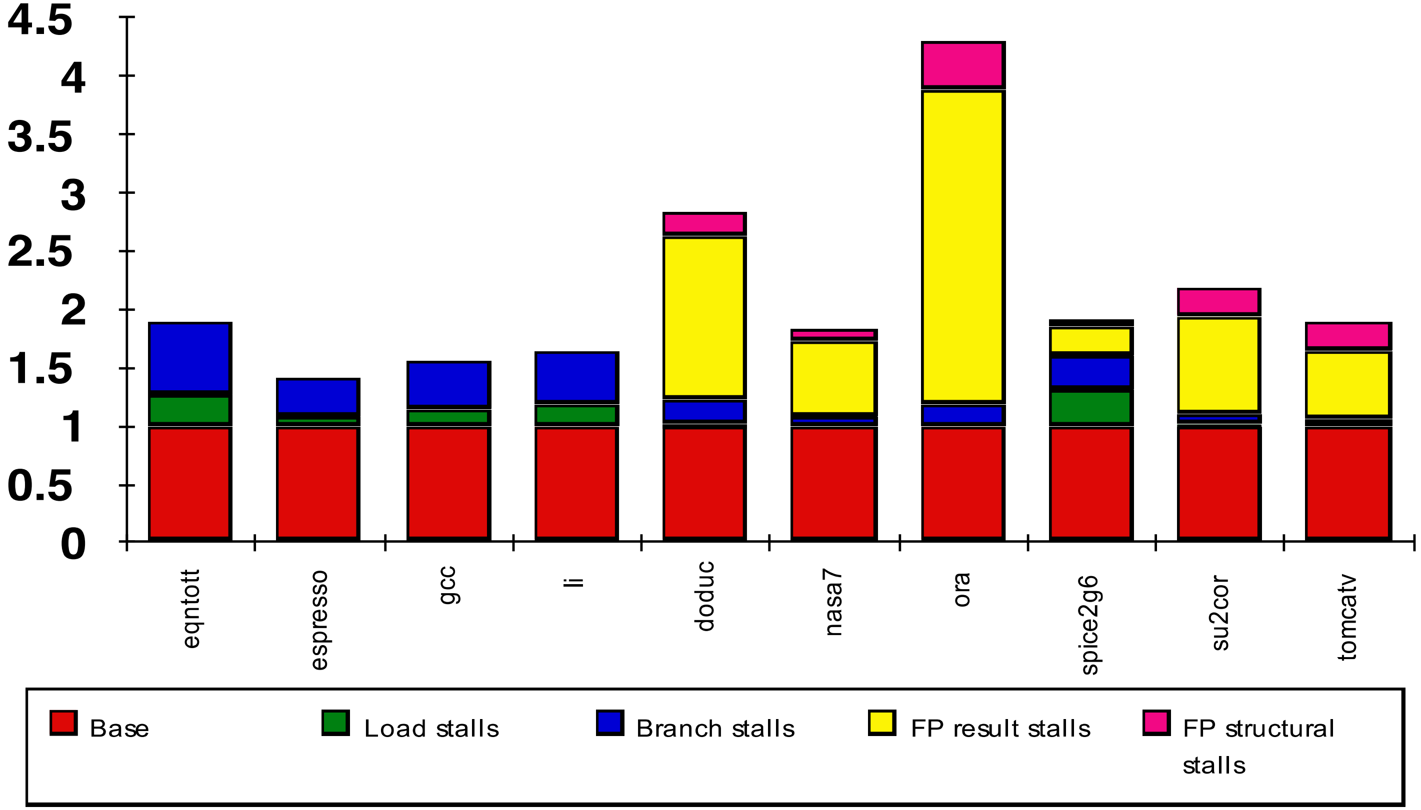
```
L.D     F0, 0(R1)

ADD.D F4, F0, F2

L.D     F0, 0(R2)
```

- **Control Dependences** : Instructions dependent via branches.

  if p1 { S1; }

R4000 Performance

- Non-Ideal CPI :
  - **Load stalls:** 1 ή 2 clock cycles
  - **Branch stalls:** 2 cycles + unfilled slots
  - **FP result stalls:** RAW data hazard (latency)
  - **FP structural stalls:** Not enough FP hardware (parallelism)



Instruction Level Parallelism (ILP)

- ILP: parallel execution of unrelated (independent) instructions

- gcc 17% control transfer instructions
  - 5 instructions + 1 branch
  - need to look beyond a code block to find more instruction level parallelism

- Loop level parallelism one opportunity
  - First SW, then HW approaches

FP Loop: where are the hazards?

```
while (R1 > 0) { M[R1] = M[R1] + F2; R1 -= 8 }
Loop:  L.D     F0,0(R1) ;F0=vector element
       ADD.D  F4,F0,F2 ;add scalar from F2
       S.D     0(R1),F4;store result
       SUBI    R1,R1,8 ;decrement pointer 8B (DW)
       BNEZ    R1,Loop ;branch R1!=zero
       NOP             ;branch delay slot
```

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	1
Integer op	Integer op	0

Stalls?

Assume 5-stage DLX (in order)

FP Loop Showing Stalls

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

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

9 cycles:

Rewrite the code to minimize stalls!

Scheduled code for FP Loop

```
1 Loop: L.D    F0,0(R1)
2       stall
3       ADD.D  F4,F0,F2
4       SUBI   R1,R1,8
5       BNEZ   R1,Loop    ;delayed branch
6       S.D    8(R1),F4   ;altered when move past SUBI
```

Move SD past BNEZ by modifying the address offset of SD

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

6 clocks: Unroll loop 4 times to make code faster?

Loop Unrolling

```
while (R1 > 0) { M[R1] = M[R1] + F2; R1 -= 8 }
```

```
while (R1 >= 4*8) {
  M[R1] = M[R1] + F2;
  M[R1-8] = M[R1-8] + F2;
  M[R1-16] = M[R1-16] + F2;
  M[R1-24] = M[R1-24] + F2;
  R1 -= 4*8
}
while (R1 > 0) { M[R1] = M[R1] + F2; R1 -= 8 }
```

Independent instructions inside the loop. Good opportunities for scheduling.

Unroll Loop 4 times: name dependencies?

```
Loop: L.D    F0,0(R1)
      ADD.D  F4,F0,F2
      S.D    0(R1),F4      ;drop SUBI & BNEZ
      L.D    F0,-8(R1)
      ADD.D  F4,F0,F2
      S.D    -8(R1),F4     ;drop SUBI & BNEZ
      L.D    F0,-16(R1)
      ADD.D  F4,F0,F2
      S.D    -16(R1),F4    ;drop SUBI & BNEZ
      L.D    F0,-24(R1)
      ADD.D  F4,F0,F2
      S.D    -24(R1),F4
      SUBI   R1,R1,#32     ;alter to 4*8
      BNEZ   R1,LOOP
      NOP
```

Unroll Loop 4 times: name dependencies?

```
1  Loop:L.D    F0,0(R1)
2       ADD.D  F4,F0,F2
3       S.D    0(R1),F4      ;drop SUBI & BNEZ
4       L.D    F0,-8(R1)
5       ADD.D  F4,F0,F2
6       S.D    -8(R1),F4     ;drop SUBI & BNEZ
7       L.D    F0,-16(R1)
8       ADD.D  F4,F0,F2
9       S.D    -16(R1),F4    ;drop SUBI & BNEZ
10      L.D    F0,-24(R1)
11      ADD.D  F4,F0,F2
12      S.D    -24(R1),F4
13      SUBI   R1,R1,#32     ;alter to 4*8
14      BNEZ   R1,LOOP
15      NOP
```

How to deal with these?

No name dependencies now!

```
Loop:L.D    F0,0(R1)
     ADD.D  F4,F0,F2
     S.D    0(R1),F4      ;drop SUBI & BNEZ
     L.D    F6,-8(R1)
     ADD.D  F8,F6,F2
     S.D    -8(R1),F8     ;drop SUBI & BNEZ
     L.D    F10,-16(R1)
     ADD.D  F12,F10,F2
     S.D    -16(R1),F12   ;drop SUBI & BNEZ
     L.D    F14,-24(R1)
     ADD.D  F16,F14,F2
     S.D    -24(R1),F16
     SUBI   R1,R1,#32     ;alter to 4*8
     BNEZ   R1,LOOP
     NOP
```

“register renaming” removed WAR/WAW stalls

Unroll Loop 4 times

```
Loop: L.D     F0,0(R1)
      ADD.D   F4,F0,F2
      S.D     0(R1),F4      ;drop SUBI & BNEZ
      L.D     F6,-8(R1)
      ADD.D   F8,F6,F2
      S.D     -8(R1),F8     ;drop SUBI & BNEZ
      L.D     F10,-16(R1)
      ADD.D   F12,F10,F2
      S.D     -16(R1),F12   ;drop SUBI & BNEZ
      L.D     F14,-24(R1)
      ADD.D   F16,F14,F2
      S.D     -24(R1),F16
      SUBI    R1,R1,#32     ;alter to 4*8
      BNEZ    R1,LOOP
      NOP
```

1 cycle stall

2 cycles stall

eliminates overhead instructions, but increases code size

Rewrite loop to minimize stalls?

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration

Assumes R1 is multiple of 4

Schedule Unrolled Loop

```
1  Loop:L.D    F0,0(R1)
2       L.D    F6,-8(R1)
3       L.D    F10,-16(R1)
4       L.D    F14,-24(R1)
5       ADD.D  F4,F0,F2
6       ADD.D  F8,F6,F2
7       ADD.D  F12,F10,F2
8       ADD.D  F16,F14,F2
9       S.D    0(R1),F4
10      S.D    -8(R1),F8
11      S.D    -16(R1),F12
12      SUBI   R1,R1,#32
13      BNEZ   R1,LOOP
14      S.D    8(R1),F16 ; 8-32 = -24
```

14 clock cycles, or 3.5 per iteration

- **What kind of assumptions did we use to reorder and move the instructions?**
  - OK to move store past SUBI even though changes register
  - **OK to move loads before stores/add: get right data?**
  - When is it safe for compiler to do such changes?

Compiler Perspectives on Code Movement

- Name Dependencies are hard to identify for Memory Accesses
  - 100(R4) == 20(R6)?
  - for different iterations of the loop, is 20(R6) == 20(R6)?
- In our example the compiler must understand that when R1 does not change then:

$$0(R1) \neq -8(R1) \neq -16(R1) \neq -24(R1)$$

- There were no dependencies between some loads and stores so they could be moved by each other

When is it safe to unroll a loop?

- Example: Are there dependencies? (A,B,C distinct & non-overlapping)

```
for (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i];     /* S1 */
      B[i+1] = B[i] + A[i+1];   /* S2 */
}
```

- `S2` uses the value, `A[i+1]`, computed by `S1` in the same iteration.
- `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 form of dependence (across iterations) is called loop-carried dependence
- In our prior example, each iteration was distinct. Dependences in the above example force successive iterations of this loop to execute in series.
- Implies that iterations can't be executed in parallel, right ?

Loop-carried dependence: No parallelism?

- Example:

```
for (i=0; i<100; i=i+1) {
        A[i]   = A[i] + B[i];   /* S1 */
        B[i+1] = C[i] + D[i];   /* S2 */
}
```

- `S1` uses the value of `B[i]` which is produced by a previous iteration (loop-carried depedence).
- There is no other dependency. Hence, this dependence is not circular. We can conclude that the loop can be parallel.

```
A[0] = A[0] + B[0]
B[1] = C[0] + D[0]
A[1] = A[1] + B[1]
B[2] = C[1] + D[1]
A[2] = A[2] + B[2]
B[3] = C[2] + D[2]
  . . .
```

```
A[0] = A[0] + B[0]
B[1] = C[0] + D[0]
A[1] = A[1] + B[1]
B[2] = C[1] + D[1]
A[2] = A[2] + B[2]
B[3] = C[2] + D[2]
  . . .
```

Loop-carried dependence: No parallelism?

- Example:

```
for (i=0; i<100; i=i+1) {
        A[i] = A[i] + B[i];     /* S1 */
        B[i+1] = C[i] + D[i];   /* S2 */
}
```

```
A[0] = A[0] + B[0];                    /* start-up code */
for (i=0; i<99; i=i+1) {
        B[i+1] = C[i] + D[i];        /* S2 */
        A[i+1] = A[i+1] + B[i+1];    /* S1 */
}
B[100] = C[99] + D[99];                /* clean-up code */
```

Recurrence – Depedence Distance

- Example:

```
for (i=1; i< 100; i=i+1) {
        Y[i] = Y[i-1] + Y[i];
}
```

  loop-carried dependence in **recurrence** form.

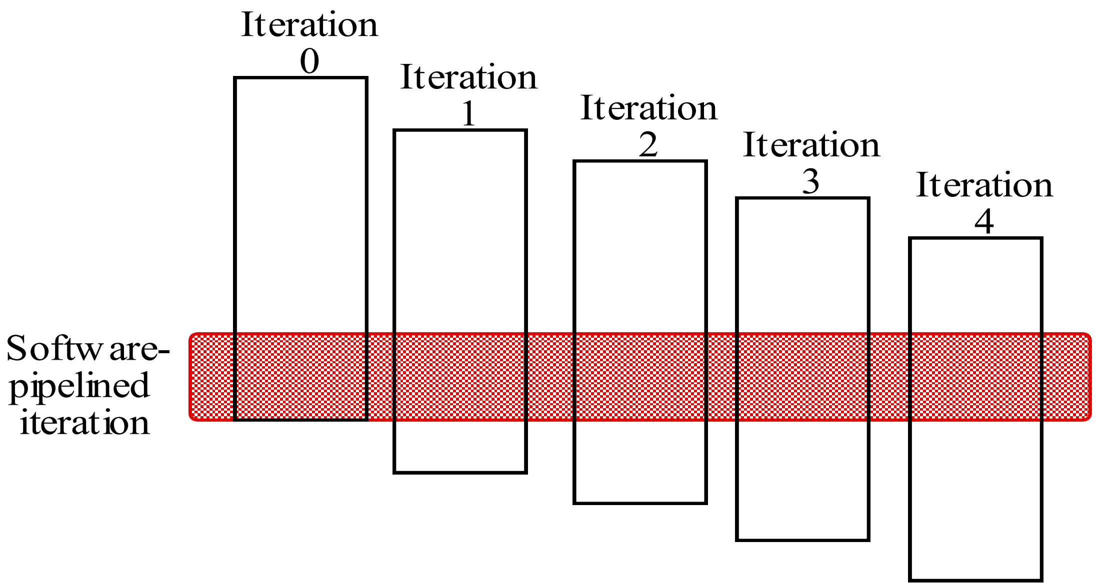
- Example:

```
for (i=5; i< 100; i=i+1) {
        Y[i] = Y[i-5] + Y[i];
}
```

  Iteration `i` depends on iteration `i-5`, thus it has a **dependence distance** of 5. The longer the dependence distance the more potential to extract parallelism.

Alternative: Software Pipelining

- Observation: If the iterations of the loop are independent, then we can exploit more ILP by executing instructions from different iterations of the loop.
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop without loop unrolling (Tomasulo in SW)



Software Pipelining: Example

Iteration 0	Iteration 1	Iteration 2	Iteration 3	Iteration 4
`LD F0,0(R1)`	*start-up code*			
`ADDD F4,F0,F2`	`LD F0,-8(R1)`			
`SD 0(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-16(R1)`		
	`SD -8(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-24(R1)`	
		`SD -16(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-32(R1)`
			`SD -24(R1),F4`	`ADDD F4,F0,F2`
			finish-up code	`SD -32(R1),F4`

Software Pipelining: Example

Iteration 0	Iteration 1	Iteration 2	Iteration 3	Iteration 4
LD F0,0(R1)	*start-up code*			
ADDD F4,F0,F2	LD F0,-8(R1)			
SD 0(R1),F4	ADDD F4,F0,F2	LD F0,-16(R1)		
	SD -8(R1),F4	ADDD F4,F0,F2	LD F0,-24(R1)	
		SD -16(R1),F4	ADDD F4,F0,F2	LD F0,-32(R1)
			SD -24(R1),F4	ADDD F4,F0,F2
			finish-up code	SD -32(R1),F4

Software Pipelining: Example

Iteration 0	Iteration 1	Iteration 2	Iteration 3	Iteration 4
`LD F0,0(R1)`	*start-up code*			
`ADDD F4,F0,F2`	`LD F0,-8(R1)`			
`SD 0(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-16(R1)`		
	`SD -8(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-24(R1)`	
		`SD -16(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-32(R1)`
			`SD -24(R1),F4`	`ADDD F4,F0,F2`
			finish-up code	`SD -32(R1),F4`

Software Pipelining: Example

Iteration 0	Iteration 1	Iteration 2	Iteration 3	Iteration 4
`LD F0,0(R1)`	*start-up code*			
`ADDD F4,F0,F2`	`LD F0,-8(R1)`			
`SD 0(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-16(R1)`		
	`SD -8(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-24(R1)`	
		`SD -16(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-32(R1)`
			`SD -24(R1),F4`	`ADDD F4,F0,F2`
			finish-up code	`SD -32(R1),F4`

Software Pipelining: Example

Iteration 0	Iteration 1	Iteration 2	Iteration 3	Iteration 4
`LD F0,0(R1)`	*start-up code*			
`ADDD F4,F0,F2`	`LD F0,-8(R1)`			
`SD 0(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-16(R1)`		
	`SD -8(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-24(R1)`	
		`SD -16(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-32(R1)`
			`SD -24(R1),F4`	`ADDD F4,F0,F2`
			finish-up code	`SD -32(R1),F4`

Software Pipelining: Example

Iteration 0	Iteration 1	Iteration 2	Iteration 3	Iteration 4
`LD F0,0(R1)`	*start-up code*			
`ADDD F4,F0,F2`	`LD F0,-8(R1)`			
`SD 0(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-16(R1)`		
	`SD -8(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-24(R1)`	
		`SD -16(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-32(R1)`
			`SD -24(R1),F4`	`ADDD F4,F0,F2`
			finish-up code	`SD -32(R1),F4`

Software Pipelining: Example

Iteration 0	Iteration 1	Iteration 2	Iteration 3	Iteration 4
`LD F0,0(R1)`	*start-up code*			
`ADDD F4,F0,F2`	`LD F0,-8(R1)`			
`SD 0(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-16(R1)`		
	`SD -8(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-24(R1)`	
		`SD -16(R1),F4`	`ADDD F4,F0,F2`	`LD F0,-32(R1)`
			`SD -24(R1),F4`	`ADDD F4,F0,F2`
			finish-up code	`SD -32(R1),F4`

Software Pipelining: Example

Before: Unroll 3 times

```
1   L.D    F0,0(R1)
2   ADD.D  F4,F0,F2
3   S.D    0(R1),F4
4   L.D    F6,-8(R1)
5   ADD.D  F8,F6,F2
6   S.D    -8(R1),F8
7   L.D    F10,-16(R1)
8   ADD.D  F12,F10,F2
9   S.D    -16(R1),F12
10  SUBI   R1,R1,#24
11  BNEZ   R1,LOOP
```

After: Software Pipelined

```
1   S.D    0(R1),F4     ; Stores M[i]
2   ADD.D  F4,F0,F2     ; Adds to M[i-1]
3   L.D    F0,-16(R1)   ; Loads M[i-2]
4   SUBI   R1,R1,#8     ; i = i - 1
5   BNEZ   R1,LOOP
```

5 cycles per iteration

RAW hazards convert to WAR hazards.

Software Pipelining vs Loop Unrolling

Symbolic Loop Unrolling

- ➢ Maximize result-use distance
- ➢ Less code space than unrolling

But..

- – Harder to implement
- – Execution of SUB & BNEZ in every iteration