

Computer Architecture

Dr. Haroon Mahmood

Assistant Professor

NUCES Lahore

CPU Performance

- For a given instruction set architecture, increases in CPU performance can come from three sources:
 1. lower the instruction count or generate instructions with a lower average CPI
 2. lower the CPI
 3. Increases in clock rate

	Instruction Count	CPI	Clock cycle time
Program	X	X	
Compiler	X	X	
Instruction Set	X	X	
Organization		X	X
Technology			X

Does doubling the clock rate
double the performance?

Amdahl's law

- The performance improvement to be gained from using some faster mode of execution is limited by the **fraction** of the time the faster mode can be used.
- This implies that the time consumed by events whose performance is not improved limits the effect of any improvement.
 - **Lowest performer restricts all others.**
- Execution Time After Improvement = Execution Time Unaffected + (Execution Time Affected / Amount of Improvement)

Example

Example: "Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?"

Answer

- Execution time after improvement = $(100 - 80 \text{ seconds}) + (80 \text{ seconds} / n)$
- Since we want the performance to be 4 times faster, the new execution time should be 25 seconds, giving:
- $25 \text{ seconds} = (20 \text{ seconds}) + (80 \text{ seconds} / n)$
 $5 = (80 \text{ seconds} / n)$
 $n = 80/5 = 16 \text{ times}$
- 5 times improvement?
- There is no amount by which we can enhance multiply to achieve a fivefold increase in performance, if multiply accounts for only 80% of the workload.

Amdahl's law....

The parameter to use in measuring the effect of Amdahl's Law is **speedup**:

$$\text{Speedup} = \frac{\text{Performance using enhancement}}{\text{Performance without using enhancement}}$$

or

$$\text{Speedup} = \frac{\text{Execution time without enhancement}}{\text{Execution time with enhancement}}$$

Speedup depends on two factors

- The fraction of the computation time in the original machine that can be converted to take advantage of the enhancement

$$\text{Fraction}_{\text{enhanced}} \leq 1$$

- The improvement gained by the enhanced execution mode

$$\text{Speedup}_{\text{enhanced}} \geq 1$$

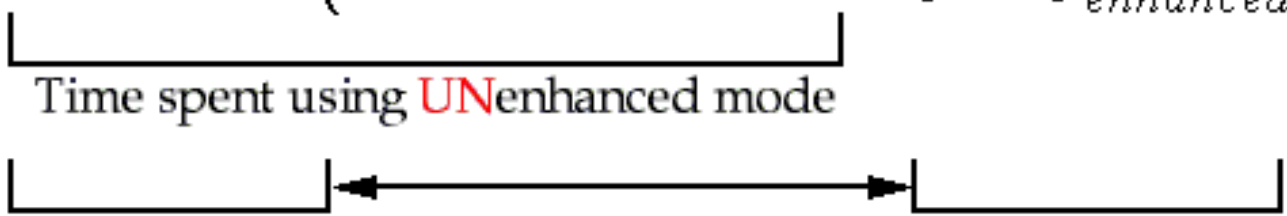
Amdahl's law

Exec time_{new} = execution time after some enhancement

Exec time_{old} = execution time before any enhancement

Fraction_{enhanced} = fraction of work using the enhancement

Speedup_{enhanced} = speedup of enhanced mode

$$\text{Exec time}_{new} = \text{Exec time}_{old} \times \left(\underbrace{(1 - \text{Fraction}_{enhanced})}_{\text{Time spent using UNenhanced mode}} + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}} \right)$$


Time spent using UNenhanced mode

Time spent using enhancement

Amdahl's law

$$\text{Speedup} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

- Suppose that we want to enhance the processor used for Web serving. The new processor is 10 times faster on computation in the Web serving application than the original processor. Assuming that the original processor is busy with computation 40 % of the time and is waiting for I/O 60 % of the time, what is the overall speedup?

$$\begin{aligned}\text{Speedup} &= 1 / (0.6 + (0.4 / 10)) \\ &= 1.56\end{aligned}$$

that is 56% speedup!!

Parallel Architectures

- **Data-Level Parallelism**
- **Task-Level Parallelism**
- **Different ways of exploiting parallelism**
 - **Instruction-Level Parallelism**
 - **Vector Architectures and GPUs**
 - **Thread level Parallelism**
 - **Request level Parallelism**

The method for exploiting parallelism

The key to higher performance in microprocessors is the ability to achieve higher degree of parallelism (fine-grain, instruction-level parallelism):

- > **pipelining** : the process of breaking down task into subtasks and executing them in different parts of processor. pipelining is mostly employed in pipelined processors.
- > **Multiple units**: process of replication of executing unit. Each unit then carry same operation on different data.

Superscalar Processors

- **Superscalar Execution**
 - How it can help
 - Issues:
 - Maintaining Sequential Semantics
 - Scheduling
 - Superscalar vs. Pipelining
- **Example: Alpha 21164 and 21064**

Sequential Semantics - Review

- Instructions appear as if they executed:
 - In the order they appear in the program
 - One after the other
- **Pipelining:** Partial Overlap of Instructions
 - Initiate one instruction per cycle
 - Subsequent instructions overlap partially
 - Commit one instruction per cycle

Superscalar vs. Pipelining

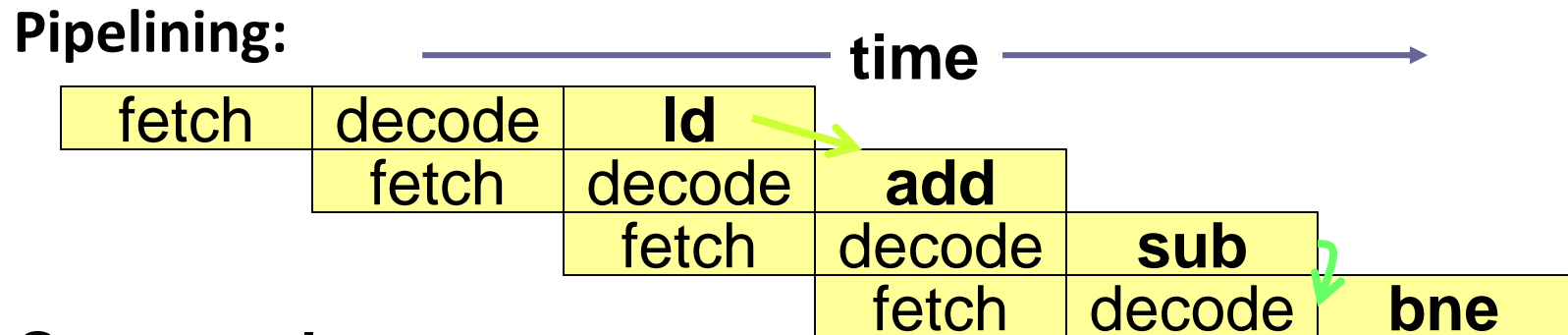
loop: **ld** r2, 10(r1)

add r3,r3, r2

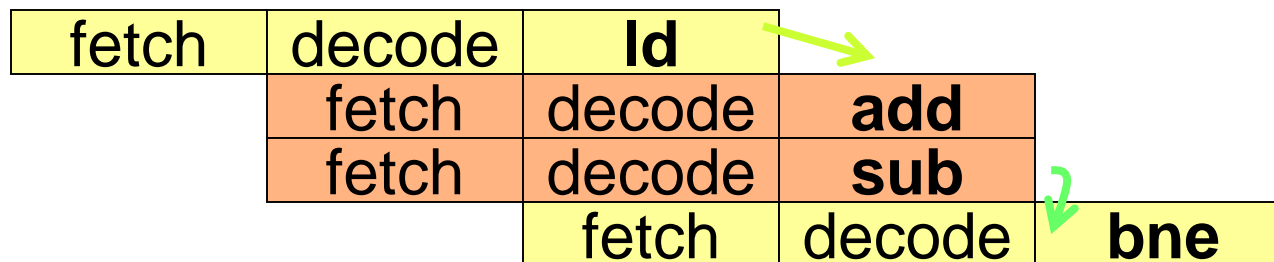
sub r1, r1, 1

bne r1, r0, loop

sum += a[i--]

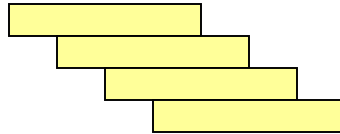


Superscalar:

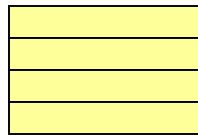


Superscalar Performance

- Performance Spectrum?
 - What if all instructions were **dependent**?
 - **Speedup = 0**, Superscalar buys us nothing



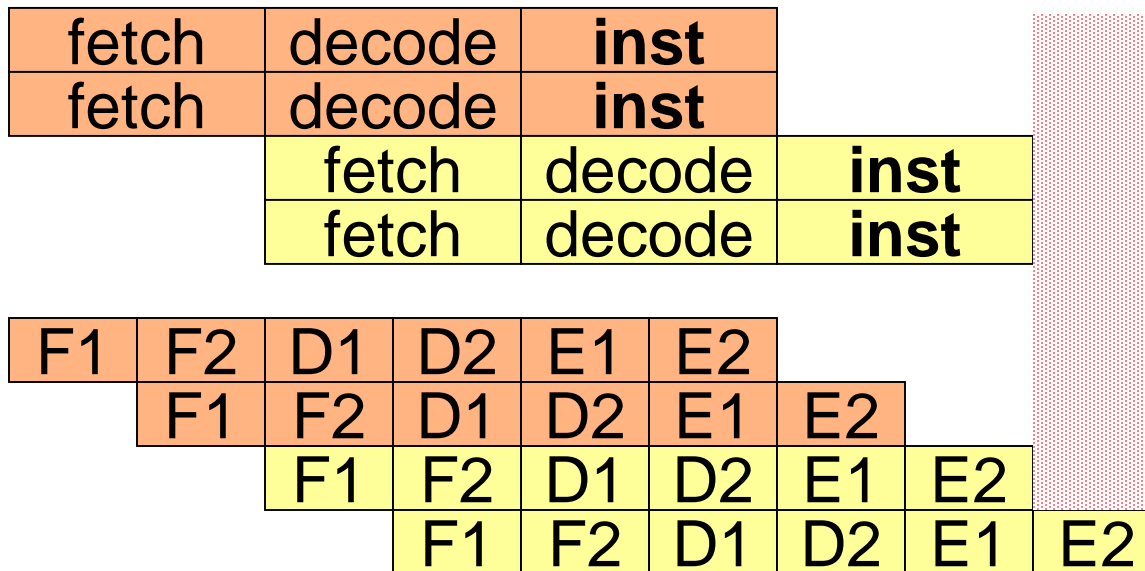
- What if all instructions were **independent**?
 - **Speedup = N** where N = superscalarity



- Again key is typical program behavior
 - Some parallelism exists

Superscalar vs. Superpipelining

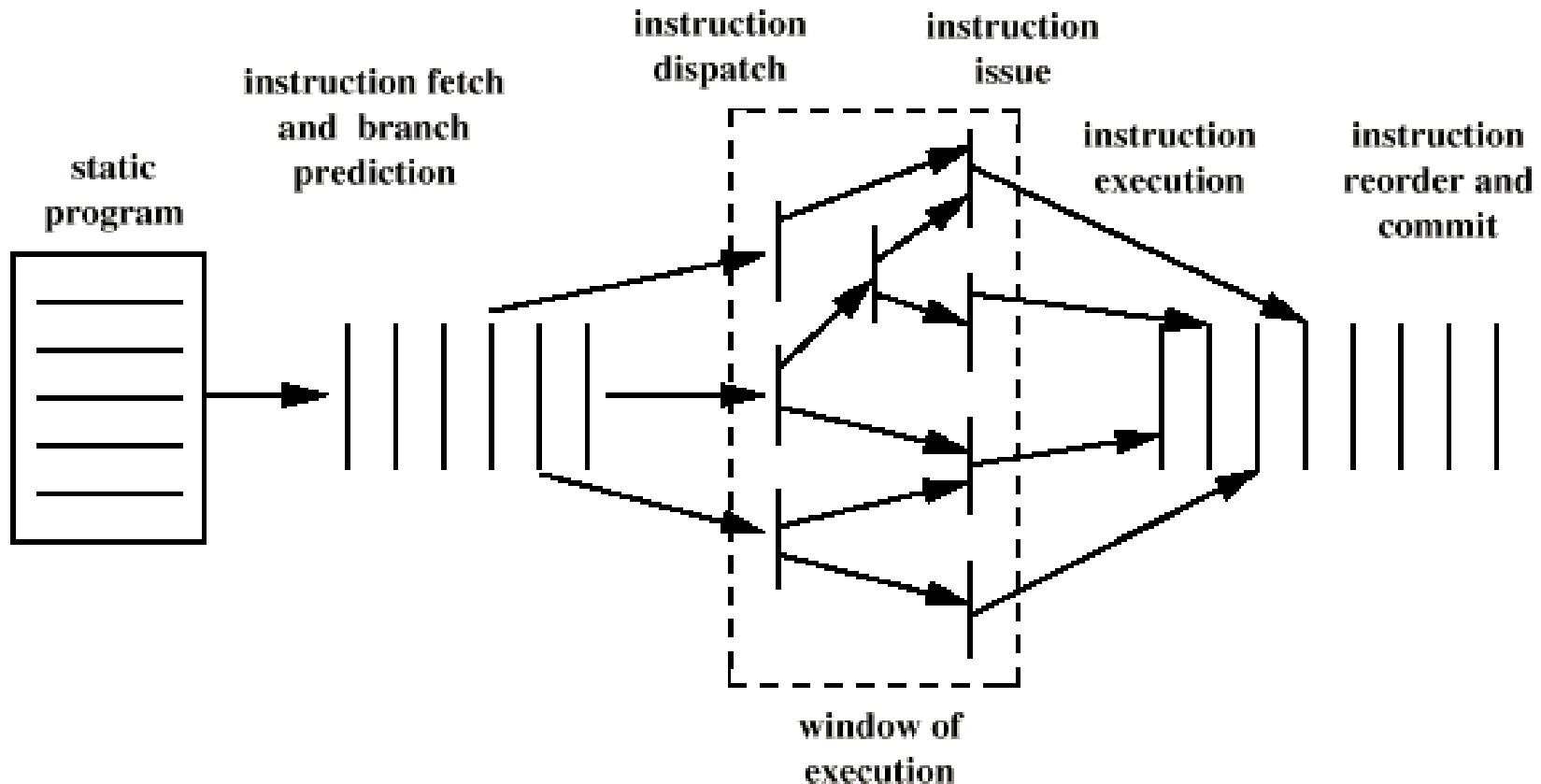
- Superpipelining:
 - *Vaguely defined as deep pipelining, i.e., lots of stages*
- Superscalar issue complexity: limits super-pipelining
- How do they compare?
 - 2-way Superscalar vs. Twice the stages
 - Not much difference.



Superscalar - In-order

- Two or more *consecutive* instructions in the original program order can execute in parallel
 - This is the dynamic execution order
- **N-way Superscalar**
 - Can issue up to N instructions per cycle
 - 2-way, 3-way, ...

Superscalar Execution



Parallel processing

Processing instructions in parallel requires three major tasks:

- 1. checking dependencies between instructions to determine which instructions can be grouped together for parallel execution;**
- 2. assigning instructions to the functional units on the hardware;**
- 3. determining when instructions are initiated placed together into a single word.**

VLIW vs Superscalar

- In VLIW and superscalar both the method pipelining and replication are employed to achieve higher performance.
- In both of them it involves specifying multiple independent operations per instruction.
- However the two architectures differ in a way they specify such instructions.
- This kind of complexity of specifying instructions in superscalar computer is at Hardware level
- While as it is software (Compiler) level in VLIW.

Static Scheduling

- **Unlike Super Scalar architectures, in the VLIW architecture all the scheduling is static**
 - **This means that they are not done at runtime by the hardware but are handled by the compiler.**
- **The compiler takes the complex instructions that need to be handled, as a result of Instruction Level Parallelism and compiles them into object code**
- **This frees up the microprocessor from having to perform the complex and continual runtime analysis that Super Scalar RISC and CISC chips must do.**

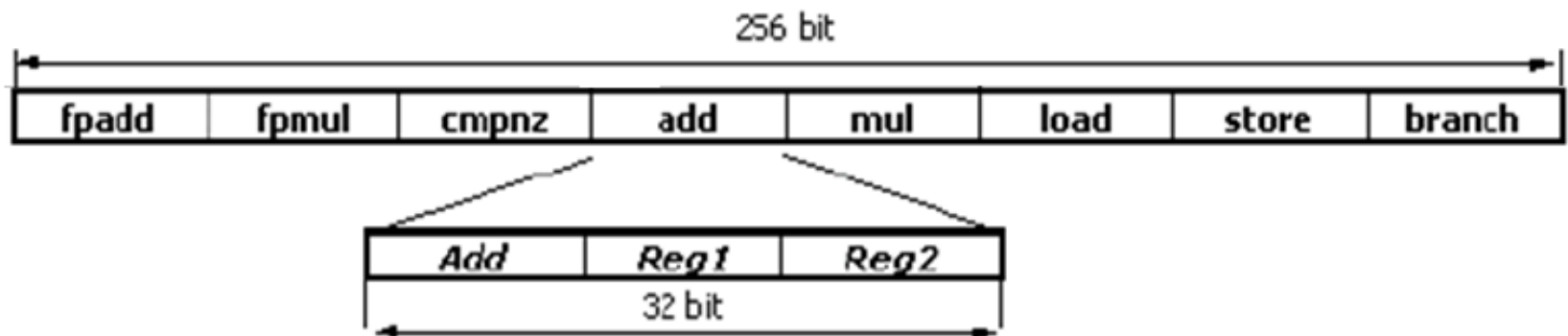
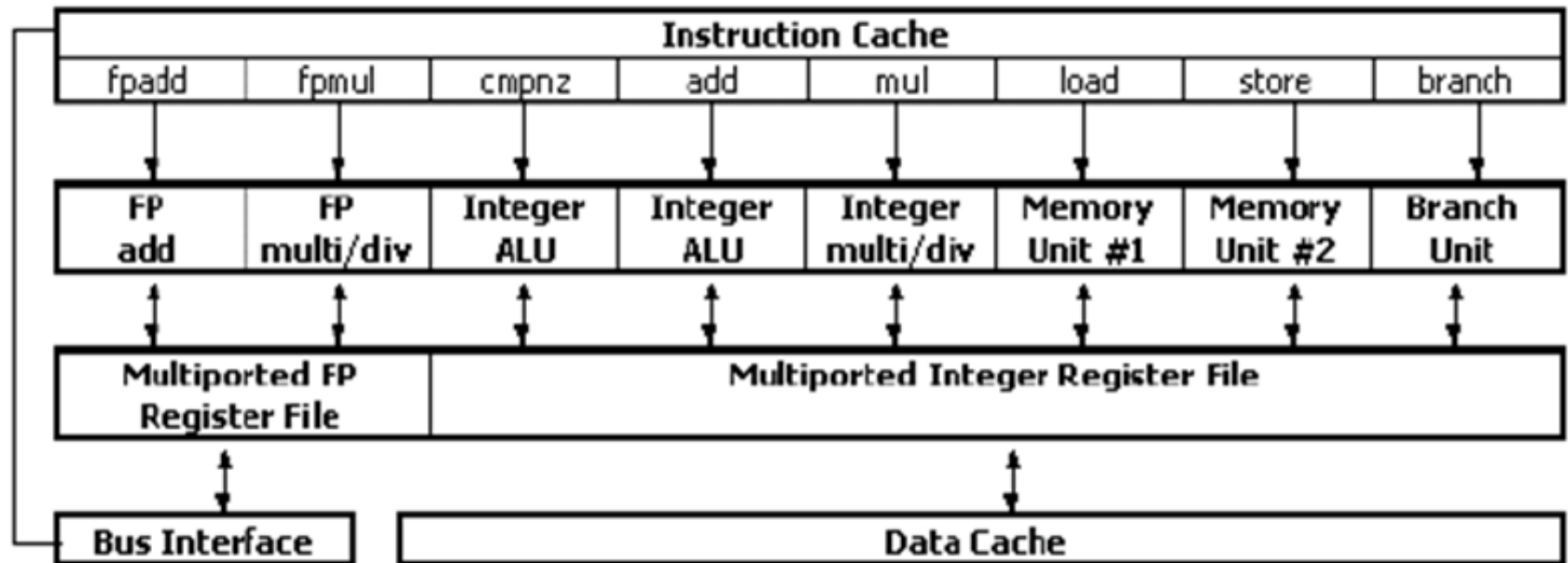
VLIW vs Super Scalar

- **Super Scalar architectures, in contrast, use dynamic scheduling that transform all ILP complexity to the hardware**
- **This leads to greater hardware complexity that is not seen in VLIW hardware**
- **VLIW chips don't need most of the complex circuitry that Super Scalar chips must use to coordinate parallel execution at runtime**

VLIW principles

- 1.The compiler analyzes dependence of all instructions among sequential code, tries to extract as much parallelism as possible.**
- 2.Based on the analysis, the compiler re-codes the piece of sequential code in VLIW instruction words.**
- 3.Finally, the work left with VLIW hardware is only fetch the VLIWs from cache, decode them, and then dispatch the independent primitive instructions to corresponding function units and execute.**

Generating of VLIW instruction words



Generating of VLIW instruction words

- 1. One VLIW instruction word contains maximum 8 primitive instructions.**
- 2. Each time, one VLIW instruction word is fetched from cache and decoded.**
- 3. After decoding, all primitive instructions in this VLIW word are issued to functional units in parallel for execution.**
- 4. These primitive instructions are from the same VLIW word, so they are guaranteed to be independent.**

Conclusion

- 1. The highly parallel implementation is much simpler and cheaper than its counterparts.**
- 2. The encoding of VLIW words implies parallelism among their primitive instructions, which results in reduced hardware complexity.**
- 3. The compiler must assemble multiple primitive instructions into a single VLIW, to make sure that multiple function units are kept busy.**

SuperScalar: In-order issue with in-order completion

- **Instruction issuing is stalled by resource conflicts, procedural or any data dependencies.**
- 1. Up to two instructions may be fetched, issued and written back at a time**
- 2. Fetch of next two instructions waits till decode buffer is cleared**
- 3. Functional units: * (2 clocks), /(2 clocks), (+,-) 1 clock.**
- 4. Data dependency stalls instruction issuing until the execution of the earlier instruction is completed.**
- 5. In RAW later instruction may be issued only after the earlier instruction has written the result.**

Superscalar execution

3 functional units: * (2 clocks), /(2 clocks), (+,-) 1 clock.

1. $R3 = R0 * R1$

2. $R4 = R0 + R2$

3. $R5 = R0 / R1$

4. $R6 = R1 + R4$

5. $R7 = R1 * R2$

6. $R1 = R0 - R2$

7. $R3 = R3 * R1$

8. $R1 = R4 + R4$

decode	/	*	+/-	write	CY
					1
					2
					3
					4
					5
					6
					7
					8
					9
					10
					11
					12
					13
					14
					15

Superscalar execution (in-order)

3 functional units: * (2 clocks), /(2 clocks), (+,-) 1 clock.

1. $R3 = R0 * R1$

2. $R4 = R0 + R2$

3. $R5 = R0 / R1$

4. $R6 = R1 + R4$

5. $R7 = R1 * R2$

6. $R1 = R0 - R2$

7. $R3 = R3 * R1$

8. $R1 = R4 + R4$

decode	/	*	+/-	write	CY
1 2					1
3 4		1 2			2
	3 1				3
	3			1 2	4
5 6			4	3	5
		5			6
		5			7
7 8			6	5	8
7 8				6	9
		7			10
		7			11
			8		12
					13
				8	14
					15

Superscalar execution

3 functional units: * (2 clocks), /(2 clocks), (+,-) 1 clock.

1. $R3 = R0 * R1$

2. $R4 = R0 + R2$

3. $R5 = R0 / R1$

4. $R6 = R1 + R4$

5. $R7 = R1 * R2$

6. $R8 = R0 - R2$

7. $R3 = R3 * R8$

8. $R9 = R4 + R4$

decode	/	*	+/-	write	CY
1 2					1
3 4		1 2			2
4	3 1				3
4	3			1 2	4
5 6			4	3	5
7 8		5 6		4	6
7		5			7
7				5 6	8
		7 8			9
		7			10
				7 8	11
					12
					13
					14
					15

Superscalar execution (out of order)

3 functional units: * (2 clocks), /(2 clocks), (+,-) 1 clock.

1. $R3 = R0 * R1$

2. $R4 = R0 + R2$

3. $R5 = R0 / R1$

4. $R6 = R1 + R4$

5. $R7 = R1 * R2$

6. $R8 = R0 - R2$

7. $R3 = R3 * R8$

8. $R9 = R4 + R4$

decode		/	*	+/-	write		CY
1	2						1
3	4		1	2			2
5	4	3	1		2		3
6	7	3	5	4	1		4
8	7		5	6	3	4	5
				8	5	6	6
			7			8	7
			7				8
					7		9
							10
							11
							12
							13
							14
							15

Loop Example

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

Source code

```
Loop:  L.D      F0, 0(R1)      ; F0 = array element  
        ADD.D   F4, F0, F2     ; add scalar  
        S.D     F4, 0(R1)     ; store result  
        DADDUI  R1, R1, # -8   ; decrement address pointer  
        BNE     R1, R2, Loop   ; branch if R1 != R2  
        NOP
```

Assembly code

Loop Example

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

Source code

LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall

```
Loop:  L.D      F0, 0(R1)      ; F0 = array element  
        ADD.D   F4, F0, F2     ; add scalar  
        S.D      F4, 0(R1)     ; store result  
        DADDUI  R1, R1, #-8     ; decrement address pointer  
        BNE     R1, R2, Loop    ; branch if R1 != R2  
        NOP
```

Assembly code

```
Loop:  L.D      F0, 0(R1)      ; F0 = array element  
        stall  
        ADD.D   F4, F0, F2     ; add scalar  
        stall  
        stall  
        S.D      F4, 0(R1)     ; store result  
        DADDUI  R1, R1, #-8     ; decrement address pointer  
        stall  
        BNE     R1, R2, Loop    ; branch if R1 != R2  
        stall
```

10-cycle
schedule

Smart Schedule

```
Loop:  L.D      F0, 0(R1)
        stall
        ADD.D   F4, F0, F2
        stall
        stall
        S.D     F4, 0(R1)
        DADDUI  R1, R1, # -8
        stall
        BNE     R1, R2, Loop
        stall
```



```
Loop:  L.D      F0, 0(R1)
        DADDUI  R1, R1, # -8
        ADD.D   F4, F0, F2
        stall
        BNE     R1, R2, Loop
        S.D     F4, 8(R1)
```

LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall

- By re-ordering instructions, it takes 6 cycles per iteration instead of 10
- We were able to violate an anti-dependence easily because an immediate was involved
- Loop overhead (instrs that do book-keeping for the loop): 2
- Actual work (the ld, add.d, and s.d): 3 instrs
- Can we somehow get execution time to be 3 cycles per iteration?

Problem 1

LD -> any : 1 stall
FPMUL -> any: 5 stalls
FPMUL -> ST : 4 stalls
IntALU -> BR : 1 stall

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

Source code

```
Loop:  L.D      F0, 0(R1)      ; F0 = array element  
        MUL.D   F4, F0, F2     ; multiply scalar  
        S.D      F4, 0(R2)     ; store result  
        DADDUI  R1, R1, #-8     ; decrement address pointer  
        DADDUI  R2, R2, #-8     ; decrement address pointer  
        BNE     R1, R3, Loop    ; branch if R1 != R3  
        NOP
```

Assembly code

- How many cycles do the default and optimized schedules take?

Problem 1

LD -> any : 1 stall
FPMUL -> any: 5 stalls
FPMUL -> ST : 4 stalls
IntALU -> BR : 1 stall

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

Source code


```
Loop:  L.D      F0, 0(R1)      ; F0 = array element  
        MUL.D   F4, F0, F2     ; multiply scalar  
        S.D      F4, 0(R2)     ; store result  
        DADDUI  R1, R1, #-8     ; decrement address pointer  
        DADDUI  R2, R2, #-8     ; decrement address pointer  
        BNE     R1, R3, Loop    ; branch if R1 != R3  
        NOP
```

Assembly code

- How many cycles do the default and optimized schedules take?
- Unoptimized: LD 1s MUL 4s SD DA DA BNE 1s -- 12 cycles
- Optimized: LD DA MUL DA 2s BNE SD -- 8 cycles

Loop Unrolling

```
Loop:  L.D      F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        BNE     R1, R2, Loop
```




LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall

```
Loop:  L.D      F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        L.D     F6, -8(R1)
        ADD.D   F8, F6, F2
        S.D     F8, -8(R1)
        L.D     F10, -16(R1)
        ADD.D   F12, F10, F2
        S.D     F12, -16(R1)
        L.D     F14, -24(R1)
        ADD.D   F16, F14, F2
        S.D     F16, -24(R1)
        DADDUI  R1, R1, #-32
        BNE     R1, R2, Loop
```

- **Loop overhead: 2 instrs; Work: 12 instrs**
- **How long will the above schedule take to complete?**

Scheduled and Unrolled Loop

```
Loop:  L.D      F0, 0(R1)
        L.D      F6, -8(R1)
        L.D      F10, -16(R1)
        L.D      F14, -24(R1)
        ADD.D    F4, F0, F2
        ADD.D    F8, F6, F2
        ADD.D    F12, F10, F2
        ADD.D    F16, F14, F2
        S.D      F4, 0(R1)
        S.D      F8, -8(R1)
        DADDUI   R1, R1, # -32
        S.D      F12, 16(R1)
        BNE     R1, R2, Loop
        S.D      F16, 8(R1)
```



LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall

- **Execution time: 14 cycles or 3.5 cycles per original iteration**

Loop Unrolling

- **Increases program size**
- **Requires more registers**
- **To unroll an n -iteration loop by degree k , we will need (n/k) iterations of the larger loop, followed by $(n \bmod k)$ iterations of the original loop**

Automating Loop Unrolling

- **Determine the dependences across iterations: in the example, we knew that loads and stores in different iterations did not conflict and could be re-ordered**
- **Determine if unrolling will help – possible only if iterations are independent**
- **Determine address offsets for different loads/stores**
- **Dependency analysis to schedule code without introducing hazards; eliminate name dependences by using additional registers**

Problem 2

LD -> any : 1 stall
FPMUL -> any: 5 stalls
FPMUL -> ST : 4 stalls
IntALU -> BR : 1 stall

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

Source code

```
Loop:  L.D      F0, 0(R1)      ; F0 = array element  
        MUL.D   F4, F0, F2     ; multiply scalar  
        S.D      F4, 0(R2)     ; store result  
        DADDUI  R1, R1, #-8     ; decrement address pointer  
        DADDUI  R2, R2, #-8     ; decrement address pointer  
        BNE     R1, R3, Loop    ; branch if R1 != R3  
        NOP
```

Assembly code

- **How many unrolls does it take to avoid stall cycles?**

Problem 2

LD -> any : 1 stall
FPMUL -> any: 5 stalls
FPMUL -> ST : 4 stalls
IntALU -> BR : 1 stall

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

Source code

```
Loop:  L.D      F0, 0(R1)      ; F0 = array element  
        MUL.D   F4, F0, F2     ; multiply scalar  
        S.D     F4, 0(R2)     ; store result  
        DADDUI  R1, R1, #-8    ; decrement address pointer  
        DADDUI  R2, R2, #-8    ; decrement address pointer  
        BNE     R1, R3, Loop   ; branch if R1 != R3  
        NOP
```

Assembly code

- How many unrolls does it take to avoid stall cycles?
- LD 1s MUL 4s SD DA DA BNE 1s
- Degree 2: LD LD MUL MUL DA DA 1s SD BNE SD
- Degree 3: LD LD LD MUL MUL MUL DA DA SD SD BNE SD
- – 12 cyc/3 iterations

Automating Loop Unrolling

- **Determine the dependences across iterations: in the example, we knew that loads and stores in different iterations did not conflict and could be re-ordered**
- **Determine if unrolling will help – possible only if iterations are independent**
- **Determine address offsets for different loads/stores**
- **Dependency analysis to schedule code without introducing hazards; eliminate name dependences by using additional registers**