ACA

Compiler Optimization and ILP
Chapter 3 and Appendix C

ILP: Instruction Level Parallelism

- Exploiting Parallelism Among Instructions
- Ability of a processor to execute multiple instructions **simultaneously** or **overlapping in time**.

Basic Block Parallelism

- A **basic block** is a straight-line code sequence:
  - No branches in except at the entry
  - No branches out except at the exit
- Parallelism within a basic block is **limited**:
  - Instructions often depend on one another
  - Overlap potential is **less than average block size**

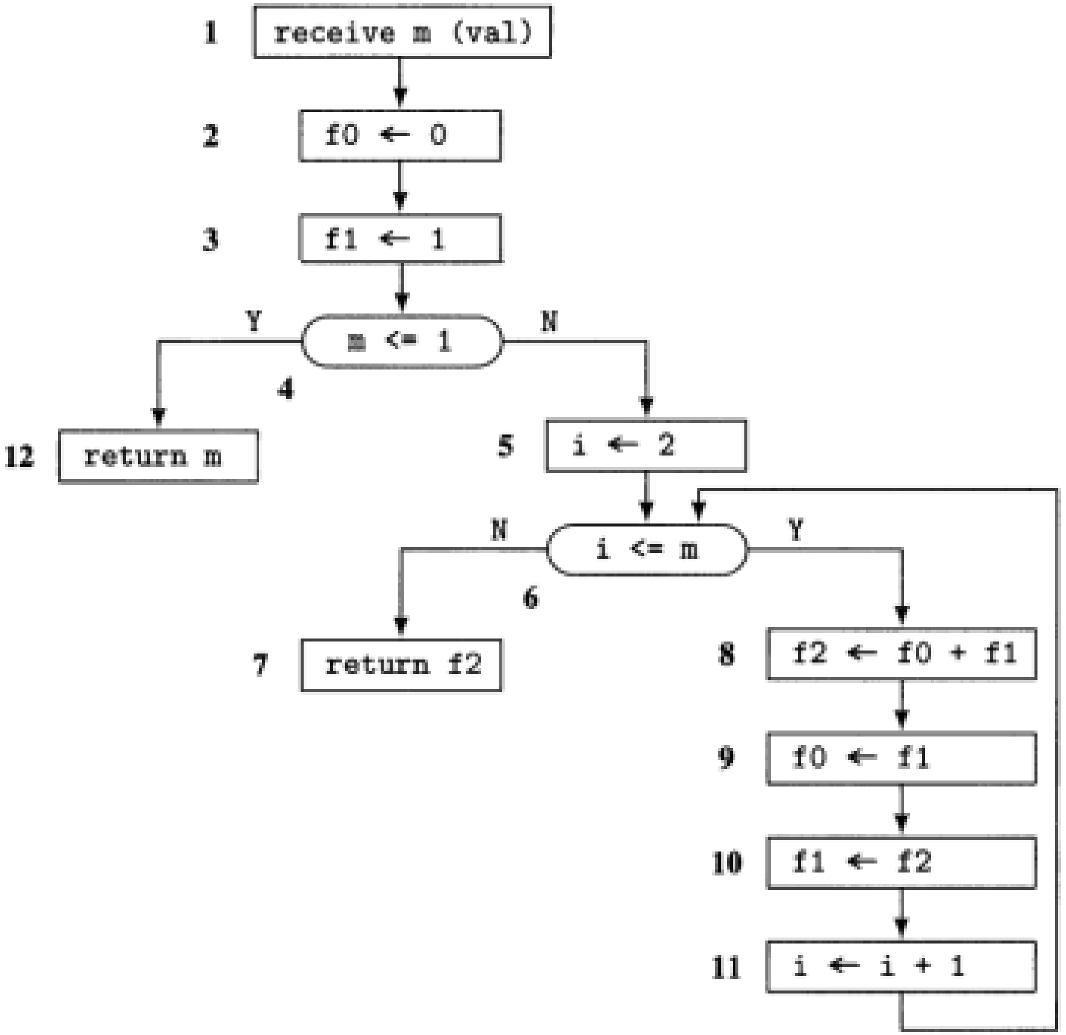
What is a Basic block?

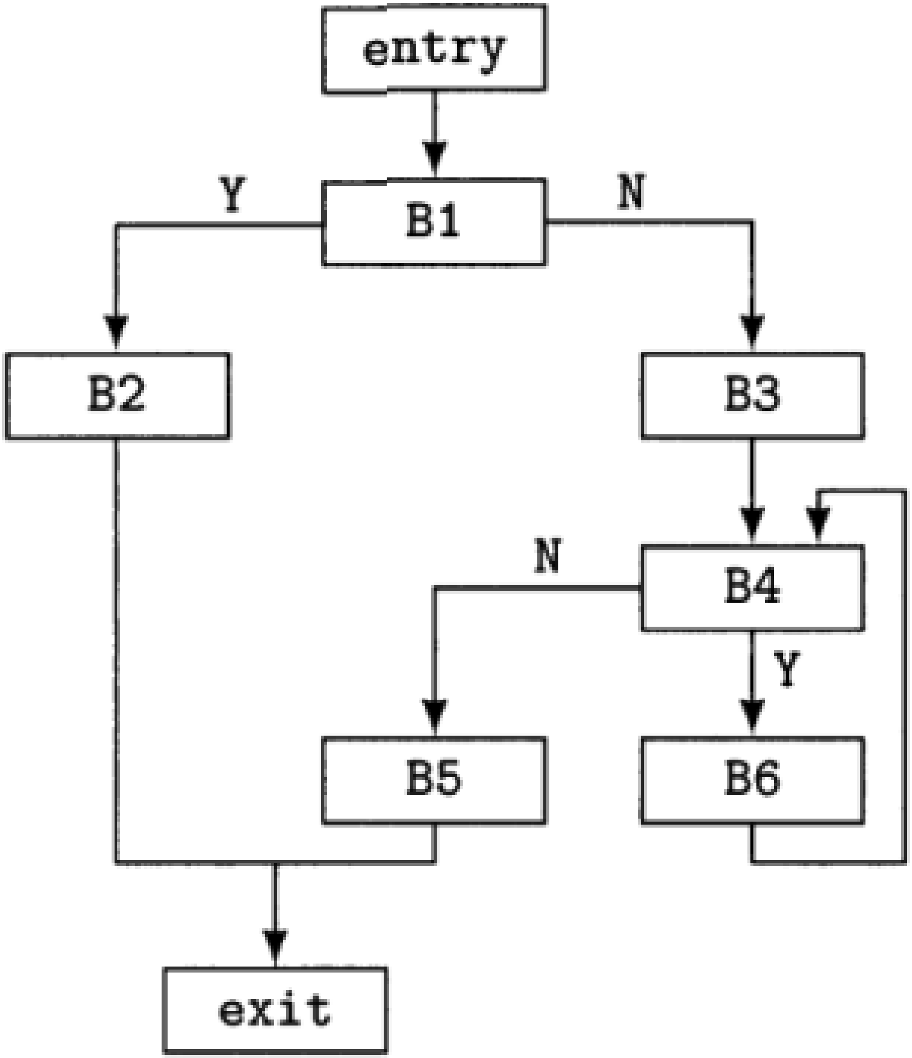
```
unsigned int fib(m)
   unsigned int m;
{  unsigned int f0 = 0, f1 = 1, f2, i;
   if (m <= 1) {
      return m;
   }
   else {
      for (i = 2; i <= m; i++) {
         f2 = f0 + f1;
         f0 = f1;
         f1 = f2;
      }
      return f2;
   }
}
```

```
        receive m (val)
        f0 ← 0
        f1 ← 1
        if m <= 1 goto L3
        i ← 2
    L1: if i <= m goto L2
        return f2
    L2: f2 ← f0 + f1
        f0 ← f1
        f1 ← f2
        i ← i + 1
        goto L1
    L3: return m
```

Source : https://www.cse.iitm.ac.in/~krishna/courses/2019/odd-cs3300/pm-lecture1.pdf

What is a Basic block?





Source : https://www.cse.iitm.ac.in/~krishna/courses/2019/odd-cs3300/pm-lecture1.pdf

Basic Block Parallelism

- A **basic block** is a straight-line code sequence:
  - No branches in except at the entry
  - No branches out except at the exit
- Parallelism within a basic block is **limited**:
  - Instructions often depend on one another
  - Overlap potential is **less than average block size**

How to increasing the ILP Beyond Basic Blocks?

- Explore Parallelism across **multiple** Basic Blocks.

Loop-Level Parallelism (LLP)

- The **simplest and most common** way to boost ILP
- Exploits parallelism **among iterations of a loop**

Within basic blocks: Limited due to dependencies
Across basic blocks: More potential, especially in loops (Loop-Level Parallelism)

Q: What are the ways of converting LLP to ILP?

Loop Unrolling - Example

```
for (int i = 0; i < 8; i++) {
    A[i] = A[i] + B[i];
}
```

- Executes 8 iterations
- Each iteration performs one addition
- Includes loop overhead: increment i, check condition, branch

Unroll this by a factor of 4

Loop Unrolling – Example (unrolling factor = 4)

```
for (int i = 0; i < 8; i += 4) {
    A[i]     = A[i]     + B[i];
    A[i + 1] = A[i + 1] + B[i + 1];
    A[i + 2] = A[i + 2] + B[i + 2];
    A[i + 3] = A[i + 3] + B[i + 3];
}
```

- Loop body is replicated 4 times
- Reduces loop control overhead (fewer branches)
- Enables better **instruction scheduling** and **parallel execution**

Ways to convert LLP to ILP

```
Consider the following example;
For (i=999; i>=0; i=i-1)
{ x[i] = x[i] + s;}
```

- Unrolling the loop:
  - Statically - by the compiler (Section 3.2 in book)
  - Dynamically by Hardware : Dynamic Scheduling, Hardware based Speculation (Section 3.5, 3.6 in book)

- Use SIMD: Vector Processors, GPUs

LLP gives repetitive instructions and independent operations across multiple iterations
=> **ILP** reorders and overlaps those to gain IPC/Performance.

FP Loop: Where are the Hazards?

Where are the stalls?

```
Consider the following example;
For (i=999; i>=0; i=i-1)
{ x[i] = x[i] + s;}
```

```
Loop:  LD    F0,0(R1)   ;F0=vector element
       ADDD  F4,F0,F2   ;add scalar from F2
       SD    F4, 0(R1)  ;store result
       SUBI  R1,R1,8    ;decrement pointer 8B (DW)
       BNEZ  R1,Loop    ;branch R1!=zero
       NOP              ;delayed branch 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	0

Figure 3.2 Latencies of FP operations used in this chapter. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is 0 because the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0 (which includes ALU operation to branch).

Branch is handled by branch delay slot.

Refresher branch-delay slot

Feature	Non-Delayed Branch	Delayed Branch
What is means?	The processor waits to **resolve the branch before executing the next instruction**. This can cause pipeline stalls and slow down performance.	The instruction immediately after the branch (in the ***delay slot***) is always executed, regardless of the branch outcome. This avoids stalls and improves efficiency if the slot is filled wisely.
Execution Model	Stall until branch resolved	Execute next instruction regardless
Pipeline Impact	Causes stalls	Avoids stalls with delay slot
Compiler Involvement	Minimal	Required for scheduling
Performance	Lower due to stalls	Higher if delay slot is used well
Used in	Common in modern CPUs	Rare in modern CPUs, used in older RISC.

FP Loop: Where are the Hazards?

```
Loop:   LD     F0,0(R1)     ;F0=vector element
        ADDD   F4,F0,F2     ;add scalar from F2
        SD     F4, 0(R1)    ;store result
        SUBI   R1,R1,8      ;decrement pointer 8B (DW)
        BNEZ   R1,Loop      ;branch R1!=zero
        NOP                 ;delayed branch 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	0
Integer op	Integer op	0

Where are the stalls?

Calculating Stalls without Optimization

```
1 Loop:LD    F0,0(R1)  ;F0=vector element
2       stall          ;1 cycle delay between Load double and FP ALU op
3       ADDD F4,F0,F2  ;add scalar in F2
4       stall
5       stall          ;2 cycles delay between FP ALU op and store
6       SD   F4,0(R1)  ;store result
7       SUBI R1,R1,8   ;decrement pointer 8B
8       BNEZ R1,Loop   ;branch R1!=zero
9       NOP stall
```

9 clocks: Rewrite code to minimize stalls?

Compiler Technique 1: Code Scheduling

Move SUBI (independent instruction) and change SD offset

```
Loop:LD    F0,0(R1)  ;F0=vector element
     SUBI  R1,R1,8   ;1 cycle delay between Load double and FP ALU op
     ADDD  F4,F0,F2  ;add scalar in F2
     stall
     stall           ;2 cycles delay between FP ALU op and store
     SD    F4,8(R1)  ; Address altered from 0(R1) to 8(R1)  when moved
 past SUBI
     BNEZ  R1,Loop   ;branch R1!=zero
     NOP stall
```

Q1. Can I reschedule the `SUBI  R1,R1,8` after `ADDD F4,F0,F2`?
Ans: YES, SUBI and ADDD are independent here

8 clocks: Rewrite code to minimize stalls?

Compiler Technique 1: Code Scheduling – Branch Delay Slot Filling

Swap BNEZ and SD by changing address of SD

```
1 Loop:LD    F0,0(R1)  ;F0=vector element
2       SUBI R1,R1,8   ;1 cycle delay between Load double and FP ALU op
3       ADDD F4,F0,F2  ;add scalar in F2
4       stall
5       stall          ;2 cycles delay between FP ALU op and store
6       SD   F4,8(R1)  ; Address altered from 0(R1) to 8(R1)  when moved
  past SUBI
7       BNEZ R1,Loop   ;branch R1!=zero
8       NOP stall
```

Q2. How will I know the offset in SD if the SUBI did not depend on an immediate value?

Ans: suppose instead of SUBI, the assembly code had SUB R1,R1,R6.. Since the value of R6 is not known statically, it is not possible to reschedule SUBI past SD.

Compiler Technique 1: Code Scheduling – Branch Delay Slot Filling

Swap BNEZ and SD

```
1 Loop:LD    F0,0(R1)  ;F0=vector element
2      SUBI  R1,R1,8   ;1 cycle delay between Load double and FP ALU op
3      ADDD  F4,F0,F2  ;add scalar in F2
4      stall
5      stall           ;2 cycles delay between FP ALU op and store
6      SD    F4,8(R1)  ; Address altered from 0(R1) to 8(R1)  when moved
 past SUBI
7      BNEZ  R1,Loop   ;branch R1!=zero
8      NOP stall
```

What assumptions made when moved code?

- OK to move store past SUBI even though changes register
- OK to move loads before stores: get right data?
- When is it safe for compiler to do such changes?

Compiler Technique 1: Code Scheduling – Branch Delay Slot Filling

Swap BNEZ and SD

```
1 Loop:LD    F0,0(R1)  ;F0=vector element
2        SUBI R1,R1,8   ;1 cycle delay between Load double and FP ALU op
3        ADDD F4,F0,F2  ;add scalar in F2
4        stall
5        stall          ;2 cycles delay between FP ALU op and store
6        SD    F4,8(R1)  ; Address altered from 0(R1) to 8(R1)  when moved
 past SUBI
7        BNEZ R1,Loop   ;branch R1!=zero
8        NOP stall
```

Q3: Can I put **SD F4,8(R1) in the branch delay slot ?** Yes
Is there any dependency on BNEZ? No

Compiler Technique 1: Code Scheduling – Branch Delay Slot Filling

Swap BNEZ and SD

```
1 Loop:LD   F0,0(R1)   ;F0=vector element
2       SUBI R1,R1,8   ;1 cycle delay between Load double and FP ALU op
3       ADDD F4,F0,F2  ;add scalar in F2
4       stall
5       BNEZ R1,Loop   ;branch R1!=zero
6       SD   F4,8(R1)  ;2 cycles delay between FP ALU op and store
 maintained, branch delay slot filled in
```

Q4: Can stall from cycle 6 be moved below/after BNEZ? Will this reshuffling alter the final results?

Ans: NO. There is **only 1 branch delay slot.**
So, swapping inst in cycle 4 and 5 will lead to SD instruction never getting executed. In BNEZ instruction branch is resolved at Decode stage(cc=6, it will jump to next address, depending on branch resolution.

Compiler Technique 1: Code Scheduling – Branch Delay Slot Filling

Swap BNEZ and SD

```
1 Loop:LD    F0,0(R1)  ;F0=vector element
2       SUBI R1,R1,8   ;1 cycle delay between Load double and FP ALU op
3       ADDD F4,F0,F2  ;add scalar in F2
4       stall
5       BNEZ R1,Loop   ;branch R1!=zero
6       SD    F4,8(R1) ;2 cycles delay between FP ALU op and store
 maintained, branch delay slot filled in
```

6 cc, Can I reduce this further?

If you **unroll** the loop by a factor of 4, how many cycles do you save?

Compiler Technique 2: Loop Unrolling - Example

```
for (int i = 0; i < 8; i++) {
    A[i] = A[i] + B[i];
}
```

- Executes 8 iterations
- Each iteration performs one addition
- Includes loop overhead: increment i, check condition, branch

Unroll this by a factor of 4

Compiler Technique 2: Loop Unrolling – Example (unrolling factor = 4)

```
for (int i = 0; i < 8; i += 4) {
    A[i]     = A[i]     + B[i];
    A[i + 1] = A[i + 1] + B[i + 1];
    A[i + 2] = A[i + 2] + B[i + 2];
    A[i + 3] = A[i + 3] + B[i + 3];
}
```

- Loop body is replicated 4 times
- Reduces loop control overhead (fewer branches)
- Enables better **instruction scheduling** and **parallel execution**

Compiler Technique 2: Loop Unrolling

- Replicates loop body multiple times
- **Purpose**:
  - Reduces frequency of branch instructions
  - Adjusts loop termination logic
  - Enables cross-iteration scheduling
- **Outcome**: More instructions per iteration → better performance

Allows compiler to:
- Reorder instructions for better pipeline utilization
- Reduce stalls by introducing **independent operations**

Compiler Technique 2: Loop Unrolling

How does this boost ILP?

- Enables more effective scheduling
- Reduces loop overhead (fewer branches and control instructions)
- Cross-iteration instruction scheduling
- Improves pipeline utilization and throughput

Compiler Technique 2: Loop Unrolling

Are there any overheads?

Increased Code Memory

- More instructions → larger binary size

Higher Register Demand

- Unrolled iterations need separate registers to avoid conflicts
- Can lead to register spilling if hardware is limited

FP Loop: Where are the Hazards?

Where are the stalls?

Consider the following example;

For (i=999; i>=0; i=i-1)

{ x[i] = x[i] + s;}

```
Loop:  LD    F0,0(R1)   ;F0=vector element
       ADDD  F4,F0,F2   ;add scalar from F2
       SD    F4, 0(R1)  ;store result
       SUBI  R1,R1,8    ;decrement pointer 8B (DW)
       BNEZ  R1,Loop    ;branch R1!=zero
       NOP              ;delayed branch 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	0

Figure 3.2 Latencies of FP operations used in this chapter. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is 0 because the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0 (which includes ALU operation to branch).

Branch is handled by branch delay slot.

Unroll factor = 4, Calculate Cycles per Iteration

Compiler Technique 2: Loop Unrolling

```
Loop: LD      F0,0(R1)
      ADDD    F4,F0,F2                   ;1 cycle delay *
      SD      0(R1),F4         ;drop SUBI & BNEZ – 2cycles delay *
      LD      F6,-8(R1)
      ADDD    F8,F6,F2                   ; 1 cycle delay
      SD      F8,-8(R1)        ;drop SUBI & BNEZ – 2 cycles delay
      LD      F10,-16(R1)
      ADDD    F12,F10,F2               ; 1 cycle delay
      SD      F12,-16(R1       ;drop SUBI & BNEZ – 2 cycles delay
      LD      F14,-24(R1)
      ADDD    F16,F14,F2               ; 1 cycle delay
      SD      F16, -24(R1)             ; 2 cycles daly
      SUBI    R1,R1,#32        ;alter to 4*8;
      BNEZ    R1,LOOP                  ; Delayed branch
      NOP                              ; Branch delay slot
```

*1 cycle delay for FP op after load. 2 cycles delay for store after FP.

14 + 4 x (1+2) + 1 = 27 clock cycles, or 6.75 cycles per iteration

Code Scheduling + Loop Unrolling

```
1  Loop: LD     F0,0(R1)
2        LD     F6,-8(R1)
3        LD     F10,-16(R1)
4        LD     F14,-24(R1)
5        ADDD   F4,F0,F2
6        ADDD   F8,F6,F2
7        ADDD   F12,F10,F2
8        ADDD   F16,F14,F2
9        SD     F4, 0(R1)
10       SD     F8, -8(R1)
11       SD     F12, -16(R1)
12       SUBI   R1,R1,#32
13       BNEZ   R1,LOOP      ; Delayed branch
14       SD     F16, 8(R1)   ; 8-32 = -24 // branch delay
slot filled
```

14 clock cycles, or 3.5 per iteration

Steps Compiler Performed to Unroll

- **1. Instruction Reordering**
  - Check if it's safe to **move the** SD **instruction** after SUBI and BNEZ
  - If safe, **adjust the offset** of SD accordingly
- **2. Loop Unrolling**
  - Analyze whether **loop iterations are independent**
  - If they are, **unrolling** can improve performance by reducing overhead
- **3. Register Renaming**
  - Rename registers to **eliminate name dependencies**
  - Prevent false dependencies that block parallel execution

Steps Compiler Performed to Unroll

- **4. Simplify Loop Control**
  - Remove **redundant test and branch instructions**
  - Adjust loop **termination and iteration logic** for clarity and efficiency
- **5. Memory Access Optimization**
  - Check if **loads and stores** from different iterations are **independent**
  - Requires analyzing memory addresses to ensure they **don't overlap**
- **6. Final Scheduling**
  - Reorder instructions while **preserving true dependencies**
  - Ensure the optimized code produces the **same result** as the original

Summary

- Loop Unrolling is essential for ILP Processors. Why?
- Limitation: Increase in Code memory and no. of registers.
- Loop unrolling is useful in a variety of processors, from simple pipelines to the **multiple-issue superscalars and VLIWs**

MORE PIPELINING

Review: 5-stage MIPS Pipeline

- Stages
  - Instruction fetch
  - Decode & read registers
  - Execute
  - Memory
  - Write-back
- Assume
  - All ALU ops in 1 cycle
  - All memory accesses in 1 cycle
  - Branches resolved in D stage

How to improve the basic pipeline?

$$Execution\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

(IC) (CPI) (Cycle Time, T_c)

- CPI = Ideal CPI + Stalls per instruction
  - Ideal CPI = ?
  - What cause stalls?
- To improve performance further we would like to decrease the CPI to less than one, but the CPI cannot be reduced below one if we issue only one instruction every clock cycle.

Improve CPI and Instruction Throughput

- **Goal**: Reduce **Cycles Per Instruction (CPI)** to **less than 1**
- **Challenge**:
  - If the processor issues **only one instruction per clock cycle**, then **CPI cannot go below 1**
- **Solution**:
  - To achieve **CPI < 1**, the CPU must execute **multiple instructions per cycle** using techniques like
    - **superscalar execution**,
    - **pipelining**, or
    - **parallelism**

What is Super Scalar Execution?

Simple Scalar ----------> Super scalar

Feature	Simple Scalar	Superscalar
Instruction Issue Rate	1 instruction per cycle	Multiple instructions per cycle
Execution Units	Single	Multiple
CPI (Cycles Per Instruction)	≥ 1, because each instruction takes at least one cycle	Can be < 1
Harware Complexity	Lower	Higher
Analogy	Simple scalar processor is like a single-lane road—only one car (instruction) can pass at a time.	Superscalar is like a multi-lane highway—several cars can move forward together

Super scalar: Key Concepts

- **Issue >1 Instruction/Cycle**
  - Enables higher throughput and lower CPI
  - Requires advanced hardware and compiler support
- **Static Scheduling by Compiler**
  - Compiler rearranges instruction sequence
  - Minimizes hazards and maximizes parallelism
- **Issue Packet of Width *n***
  - Fetch and attempt to issue up to *n* instructions per cycle
  - Known as the **issue width**

Super scalar: Key Concepts

- **Challenges in Issue Stage**
- **Hazard Detection Complexity**
  - Must check for data, control, and structural hazards
  - Includes both current and earlier instructions in execution
- **Timing Constraints**
  - Issue checks often **too complex** for a single clock cycle
  - May require multi-cycle or staged issue logic
- **Architectural Implications**
  - Behaves like an **n-fold pipeline**
  - Needs **complex issue logic** and a **large set of bypass paths** for forwarding data

Super scalar: Behaves like an *n*-fold pipeline

- This means the processor can issue and execute **n instructions simultaneously** in a single clock cycle.
- It mimics having **n parallel pipelines**, each handling one instruction.
- Example, a 2-fold pipeline can process 2 instructions at once, increasing throughput but also adding complexity.

Type	*Pipe*	*Stages*						
Int. instruction		IF	ID	EX	MEM	WB		
FP instruction		IF	ID	EX	MEM	WB		
Int. instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	MEM	WB	
Int. instruction				IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	MEM	WB

Super scalar: Needs complex issue logic

- When multiple instructions are issued together, the processor must decide **which ones can be executed in parallel** without causing errors.
- This requires **advanced scheduling logic** to:
  - Detect **data dependencies** between instructions.
  - Avoid **hazards** (like read-after-write conflicts).
  - Maintain **correct execution order**.
- The more instructions you issue per cycle, the more sophisticated this logic needs to be.

Type	Pipe	Stages						
Int. instruction		IF	ID	EX	MEM	WB		
FP instruction		IF	ID	EX	MEM	WB		
Int. instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	MEM	WB	
Int. instruction				IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	MEM	WB

Super scalar: Large set of bypass paths for forwarding data

- In pipelined execution, later instructions often need results from earlier ones.
- Instead of waiting for data to be written back to registers, **bypass paths** allow data to be forwarded directly between pipeline stages.
- With multiple instructions in flight, the processor needs **many such paths** to ensure fast and correct data delivery — especially in **out-of-order execution** or **deep pipelines**.

Type	*Pipe*	*Stages*						
Int. instruction		IF	ID	EX	MEM	WB		
FP instruction		IF	ID	EX	MEM	WB		
Int. instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	MEM	WB	
Int. instruction				IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	MEM	WB

Solve the same example for Super-Scalar

Where are the stalls?

```
Consider the following example;
For (i=999; i>=0; i=i-1)
{ x[i] = x[i] + s;}
```

```
Loop:   LD      F0,0(R1)     ;F0=vector element
        ADDD    F4,F0,F2     ;add scalar from F2
        SD      F4, 0(R1)    ;store result
        SUBI    R1,R1,8      ;decrement pointer 8B (DW)
        BNEZ    R1,Loop      ;branch R1!=zero
        NOP                  ;delayed branch 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	0

Figure 3.2 Latencies of FP operations used in this chapter. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is 0 because the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0 (which includes ALU operation to branch).

Branch is handled by branch delay slot.

2-Issue without Loop Unrolling

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

MIPS code:

```
Loop:  LD     F0,0(R1)      ;F0=vector element
       ADDD   F4,F0,F2      ;add scalar from F2
       SD     F4, 0(R1)     ;store result
       SUBI   R1,R1,8.      ;decrement pointer 8B (DW)
       BNEZ   R1,Loop       ;branch R1!=zero
```

What if executed on a 2-issue superscalar machine with one integer (EX or Ld/St) unit and one FPU?

Q: Show the execution order and calculate the Cycles per Iteration.
(i) Without Unrolling
(ii) With Unrolling factor 4,
(iii) Determine minimum unrolling factor to eliminate ALL STALLS in the pipeline?

2-Issue without Loop Unrolling

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

MIPS code:

```
Loop:  LD     F0,0(R1)      ;F0=vector element
       ADDD   F4,F0,F2      ;add scalar from F2
       SD     F4, 0(R1)     ;store result
       SUBI   R1,R1,8.      ;decrement pointer 8B (DW)
       BNEZ   R1,Loop       ;branch R1!=zero
```

What if executed on a 2-issue superscalar machine with one integer (EX or Ld/St) unit and one FPU?

=> This implies that 2 ALU exists. Int Unit for Int/LD/ST operations and FP Unit for FP operations

	EX Unit	FPU Unit
1	Loop: LD F0,0(R1)	
2	SUBI R1,R1,8	Stall – Delay between LD and FP operation
3	Stall --- Delay between ADD and SD	ADDD F4,F0,F2
4	Stall ---- Delay between ADD and SD	
5	BNEZ R1,Loop	
6	SD F4, 8(R1) - Branch Delay slot	

No improvement even with 2-issue superscalar!

Show the execution order and calculate the Cycles per Iteration. Use unrolling factor = 4

Loop Unrolling on 2-issue Superscalar (Unrolled 4 times)

EX: One Int unit and one FP ALU

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

```
1  Loop: LD    F0,0(R1)
2        LD    F6,-8(R1)
3        LD    F10,-16(R1)          ADDD  F4,F0,F2
4        LD    F14,-24(R1)          ADDD  F8,F6,F2
5        Stall                      ADDD  F12,F10,F2
6        SD    F4, 0(R1)            ADDD  F16,F14,F2
7        SD    F8, -8(R1)
8        SD    F12, -16(R1)
9        SUBI  R1,R1,#32
10       BNEZ  R1,LOOP     ; Delayed branch
11       SD    F16, 8(R1)    ; R1 was reduced by 32.
                               So, 8-32 = -24
```

11 clock cycles, or 2.75 per iteration

Q: How many times do you need to unroll (minimum unrolling factor) inorder to remove ALL STALLS in the pipeline?
Show the execution order and calculate the Cycles per Iteration

Loop Unrolling on 2-issue Superscalar (Unrolled 4 times)

- Here the **critical resource** is Int Unit. i.e. the unit which has the slowest running instruction due to dependency.
- For eliminating all stalls in the pipeline, we will try to identify the **ALU/resource in the critical path** i.e. critical resource and make sure that there are no stalls in there.

Loop Unrolling on 2-issue Superscalar (Unrolled 4 times)

EX: One Int unit and one FP ALU

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

```
1  Loop: LD    F0,0(R1)
2        LD    F6,-8(R1)
3        LD    F10,-16(R1)                 ADDD  F4,F0,F2
4        LD    F14,-24(R1)                 ADDD  F8,F6,F2
5        Stall                             ADDD  F12,F10,F2
6        SD    F4, 0(R1)                   ADDD  F16,F14,F2
7        SD    F8, -8(R1)
8        SD    F12, -16(R1)
9        SUBI  R1,R1,#32
10       BNEZ  R1,LOOP    ; Delayed branch
11       SD    F16, 8(R1)    ; R1 was reduced by 32.
                               So, 8-32 = -24
```

With efficient code scheduling, ALL stalls can be removed from the critical resource pipeline.

- Moving SUBI to cycle 5 will result in 10 clock cycles, or 2.5 per iteration

Loop Unrolling on 2-issue Superscalar (Unrolled 5 times)

	Integer instruction	FP instruction	Clock cycle
Loop:	LD F0,0(R1)		1
	LD F6,-8(R1)		2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD F8, -8 (R1)	ADDD F20,F18,F2	7
	SD F12, -16 (R1)		8
	SD F16, -24 (R1)		9
	SUBI R1,R1,#40		10
	BNEZ R1,LOOP		11
	SD F20, -32 (R1)		12

<u>LD to ADDD:</u> 1 Cycle
<u>ADDD to SD:</u> 2 Cycles

Unrolled 5 times to avoid delays

12 clocks, or 2.4 clocks per iteration (1.5X)

Solve the same example for multi-ALU Super-Scalar

Where are the stalls?

```
Consider the following example;
For (i=999; i>=0; i=i-1)
{ x[i] = x[i] + s;}
```

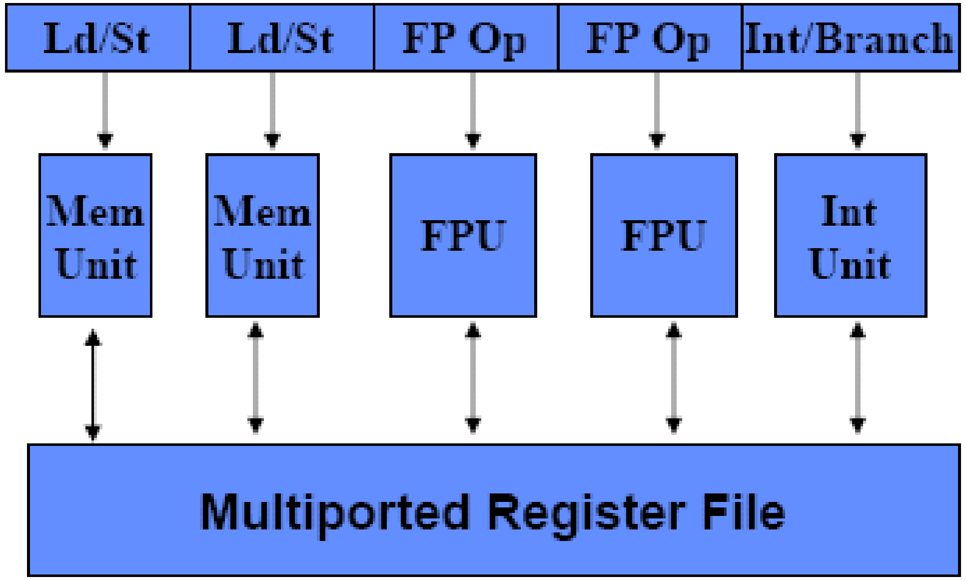
```
Loop:   LD      F0,0(R1)     ;F0=vector element
        ADDD    F4,F0,F2     ;add scalar from F2
        SD      F4, 0(R1)    ;store result
        SUBI    R1,R1,8      ;decrement pointer 8B (DW)
        BNEZ    R1,Loop      ;branch R1!=zero
        NOP                  ;delayed branch slot
```

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	0

Figure 3.2 Latencies of FP operations used in this chapter. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is 0 because the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0 (which includes ALU operation to branch).

Branch is handled by branch delay slot.

Multiple ALU units Super Scalar Hardware



<u>LD to ADDD:</u> 1 Cycle
<u>ADDD to SD:</u> 2 Cycles

MIPS code:

```
Loop:  LD      F0,0(R1)      ;F0=vector element
       ADDD  F4,F0,F2        ;add scalar from F2
       SD    F4, 0(R1),      ;store result
       SUBI  R1,R1,8         ;decrement pointer 8B (DW)
       BNEZ R1,Loop          ;branch R1!=zero
```

How many times should the loop be unrolled to avoid eliminate ALL STALLS in the pipeline?

Multiple ALU units Super Scalar Hardware

- How to identify the critical resource here?

cycle #	LD/ST Unit 1	LD/ST Unit 2	FP Unit 1	FP Unit 2	Int Unit
1	LD_1				
2					
3			$ADDD_1$		
4					SUBI
5					BNEZ
6	SD_1				
7					

$LD_{i.}$ -→ ADDDi : 1 interleaving cycle

$ADDD_{i.}$ -→ SDi : 2 interleaving cycle

- $LD_{i.}$ Denotes the LD in iteration "i"
- $SD_{1.}$ cannot be issued before cycle 6 due to dependency => critical resouce is LD/ST Unit 1
  Even though LD/ST Unit 1 and LD/ST Unit 2 are homogeneous, we typically issue the instructions in round robin manner, so busiest unit is determined as LD/ST Unit 1

Multiple ALU units Super Scalar Hardware

- How to know minimum unrolling factor for eliminating all stalls?

cycle #	LD/ST Unit 1	LD/ST Unit 2	FP Unit 1	FP Unit 2	Int Unit
1	LD_1				
2					
3			$ADDD_1$		
4					SUBI
5					BNEZ
6	SD_1				
7					

$LD_{i.}$ -→ ADDDi : 1 interleaving cycle

$ADDD_{i.}$ -→ SDi : 2 interleaving cycle

- $SD_{1.}$ cannot be issued before cycle 6 due to dependency
  => critical resouce is LD/ST Unit 1
- Need minimum 9 LDs to fill in all the stall cycles in cycle 2,3,4 and 5.

Multiple ALU units Super Scalar Hardware

- How to know minimum unrolling factor for eliminating all stalls?

cycle #	LD/ST Unit 1	LD/ST Unit 2	FP Unit 1	FP Unit 2	Int Unit
1	LD_1	LD_2			
2	LD_3	LD_4			
3	LD_5	LD_6	$ADDD_1$		
4	LD_7	LD_8			SUBI
5	LD_9				BNEZ
6	SD_1				
7					

- $SD_{1.}$ cannot be issued before cycle 6 due to dependency
  => critical resouce is LD/ST Unit 1
- Need minimum 9 LDs to fill in all the stall cycles in cycle 2,3,4 and 5.

Multiple ALU units Super Scalar Hardware

- How to know minimum unrolling factor for eliminating all stalls?

cycle #	LD/ST Unit 1	LD/ST Unit 2	FP Unit 1	FP Unit 2	Int Unit
1	LD_1	LD_2			
2	LD_3	LD_4			
3	LD_5	LD_6	$ADDD_1$		
4	LD_7	LD_8			SUBI
5	LD_9				BNEZ
6	SD_1				
7					

Q: minimum unrolling factor for eliminating all stalls? => Unrolling factor is 9

Q: minimum unrolling factor for eliminating all stalls as well as get best cycles per iteration? => Unrolling factor is 10

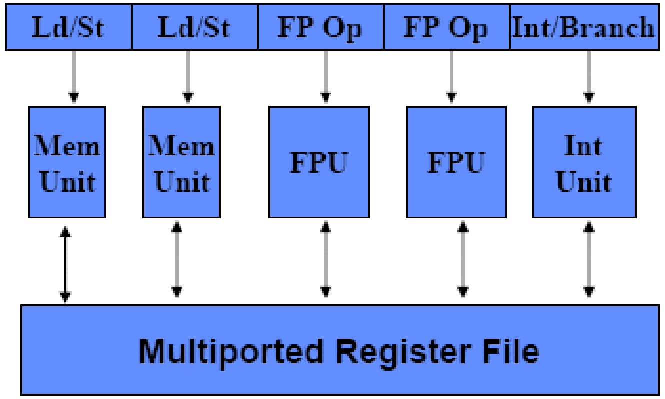
- as both unrolling 9 or 10 times takes same cycles in the unrolled loop.

Loop Unrolling in Multi-ALU units Super Scalar

Hardware in Previous Slide: 2 LD/ST, 2 FPU, 1 INT

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				2
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		3
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
		ADD.D F20,F18,F2	ADD.D F24,F22,F2		5
S.D F4, 0(R1)	S.D -8(R1),F8	ADD.D F28,F26,F2			6
S.D F12, -16(R1)	S.D -24(R1),F16				7
S.D F20, -32(R1)	S.D -40(R1),F24			DSUBUI R1,R1,#48	8
S.D F28, -0(R1)			BNEZ R1,LOOP		9
				Branch Delay slot.	10



Unrolled 7 times to avoid delays

7 iterations in 10 clocks, or 1.4 clocks per iteration

- 1.96X faster compared to previous h/w (1 integer or Ld/St) unit + 1 FPU
- 5 ALUs vs 2 ALUs in previous h/w

Note: Need more registers

Loop Unrolling in Multi-ALU units Super Scalar

Hardware in Previous Slide: 2 LD/ST, 2 FPU, 1 INT

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				2
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		3
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
STALL		ADD.D F20,F18,F2	ADD.D F24,F22,F2		5
S.D F4, 0(R1)	S.D -8(R1),F8	ADD.D F28,F26,F2			6
S.D F12, -16(R1)	S.D -24(R1),F16				7
S.D F20, -32(R1)	S.D -40(R1),F24			DSUBUI R1,R1,#48	8
S.D F28, -0(R1)				BNEZ R1,LOOP	9
				Branch Delay slot.	10

How many times should the loop be unrolled to avoid eliminate ALL STALLS in the pipeline?

Unrolling factor of 7 is incorrect as there is stall in LD/ST Unit 1 pipeline.