



# Computer Architecture *and* Organization

Lecture Series for VIT, Chennai

*Jaiber John, Intel*  
2022



# Contents

1. Module 3: Parallel Computers
  - a. Instruction Level Parallelism (ILP)
  - b. Data Level Parallelism (DLP)
  - c. Thread Level Parallelism (TLP)
  - d. Parallel Processing

<b>Module:3</b>	<b>Parallel Computers</b>	<b>8 hours</b>
Instruction Level Parallelism(ILP), Compiler Techniques for ILP & Branch Prediction, Thread Level Parallelism (TLP), Threading Concepts, Shared Memory, Message Passing, Vectorization		



# Reference Materials

1. [**Stallings2016**] Computer Organization and Architecture, William Stallings, 10th edition, 2016
2. [**Hennessy2012**] Computer Architecture, A Quantitative Approach, Fifth Edition, John L. Hennessy, David A. Patterson, 5th edition, 2012
3. [Denis2020] Performance Analysis and Tuning on Modern CPUs, 2020 Edition



# Instruction Level Parallelism (ILP)

1. Stallings2016 - ch. 16
2. Hennessy2012 - ch. 3

starvation  
anitdependency  
cpi  
war  
loop unrolling  
clock  
dependence  
branch prediction  
raw  
pipelining  
dynamic scheduling  
speculative execution  
ooo  
scheduling  
waw  
mips  
superscalar



# ILP Overview

## What is ILP?

Overlapped or parallel execution of instructions in a processor

## Why is ILP needed?

To achieve better performance (i.e reduce CPI) at the same frequency

## What are the types of ILP implementations?

1. Hardware (dynamic, at run-time.  
Ex. pipelining)
2. Software (static, at compile-time)  
Ex. Loop unrolling)

## What are the ILP techniques?

1. Instruction Pipelining
2. Superscalar Execution
3. Out of order Execution
4. Register Renaming
5. Speculative Execution
6. Branch Prediction
7. Dynamic Scheduling

## What are the constraints to achieve ILP?

- Data Dependencies
  - True Data Dependency
  - Name Dependency
  - Control Dependency (Procedural dependency)
- Data Hazards
  - Read after Write (RAW)
  - Write after Write (WAW)
  - Write after Read (WAR)

# ILP - Learning Objectives

- Understand design issues involved in ILP
- Identify various constraints and hazards in code snippets
- Implement ILP techniques to improve performance
- Understand the ILP limitations and trends
- Solve problems like these below..

**16.3** Consider the following assembly language program:

```
I1: Move R3, R7          /R3 ← (R7) /  
I2: Load R8, (R3)       /R8 ← Memory (R3) /  
I3: Add R3, R3, 4        /R3 ← (R3) + 4 /  
I4: Load R9, (R3)       /R9 ← Memory (R3) /  
I5: BLE R8, R9, L3       /Branch if (R9) > (R8) /
```

This program includes WAW, RAW, and WAR dependencies. Show these.

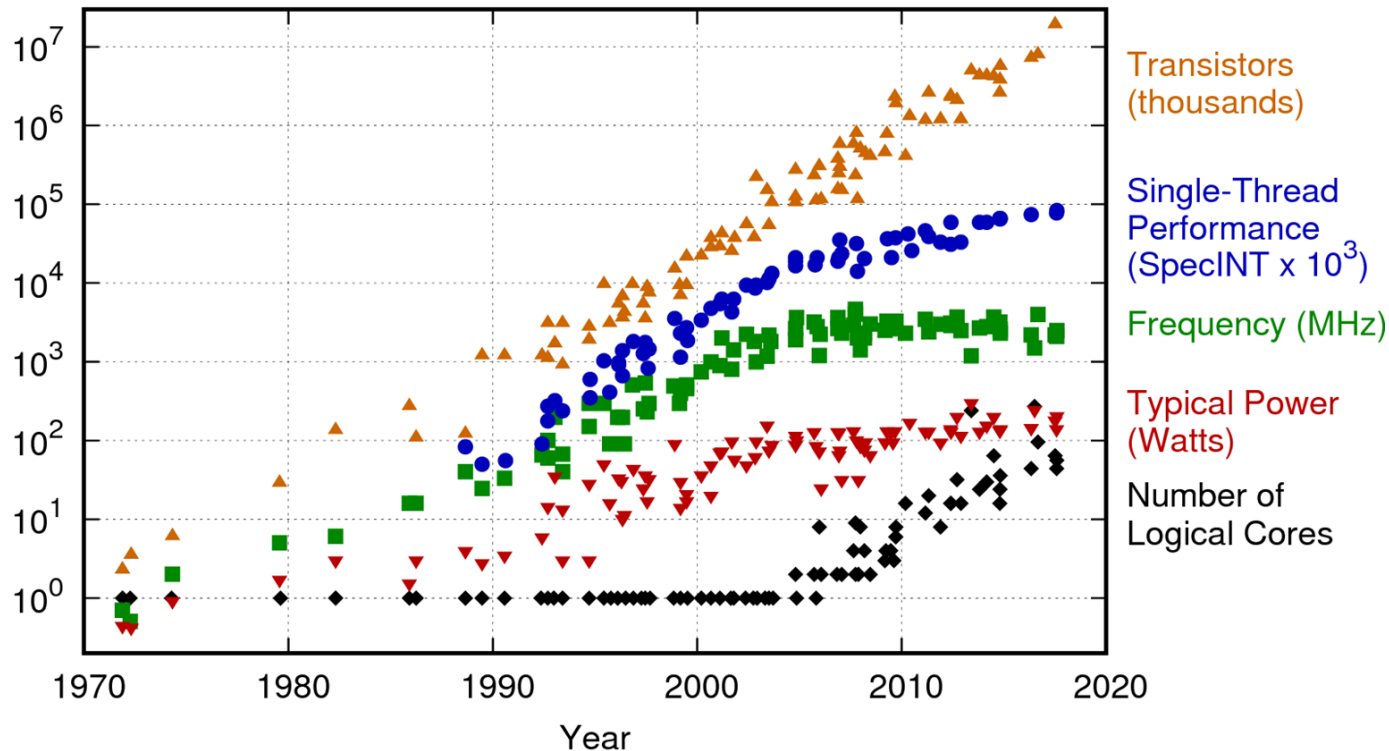
Consider the following program to be executed on this processor:

```
I1: Load R1, A          /R1 ← Memory (A) /  
I2: Add R2, R1           /R2 ← (R2) + R(1) /  
I3: Add R3, R4           /R3 ← (R3) + R(4) /  
I4: Mul R4, R5           /R4 ← (R4) + R(5) /  
I5: Comp R6              /R6 ← (R6) /  
I6: Mul R6, R7           /R6 ← (R6) × R(7) /
```

- a. What dependencies exist in the program?
- b. Show the pipeline activity for this program on the processor of Figure 16.15 using in-order issue with in-order completion policies and using a presentation similar to Figure 16.2.
- b. Repeat for in-order issue with out-of-order completion.
- c. Repeat for out-of-order issue with out-of-order completion.

# CPU Performance Trend

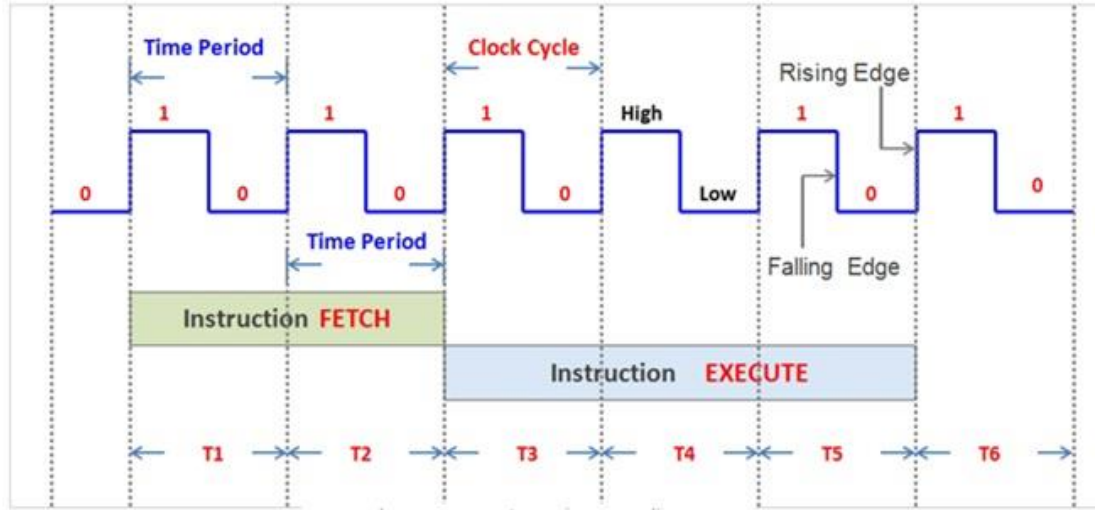
42 Years of Microprocessor Trend Data



- Transistor density doubles every 2 years (Moore's law)
- Processor frequency will remain stable
- Power usage should reduce
- Logical cores count will continue to increase
- How to increase single threaded performance?

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2017 by K. Rupp

# CPI and Performance Metrics



Simplified instruction execution cycle

Cycles Per Instruction

$$CPI = \frac{\sum_{i=1}^n (CPI_i \times I_i)}{I_c}$$

Program Execution Time

$$T = I_c \times CPI \times \tau$$
$$\tau = 1/f.$$

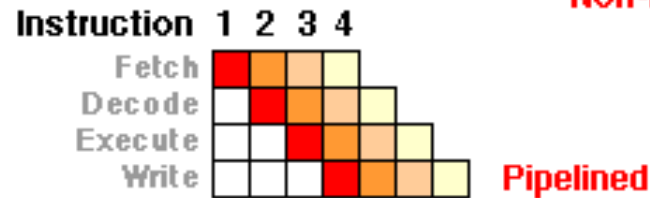
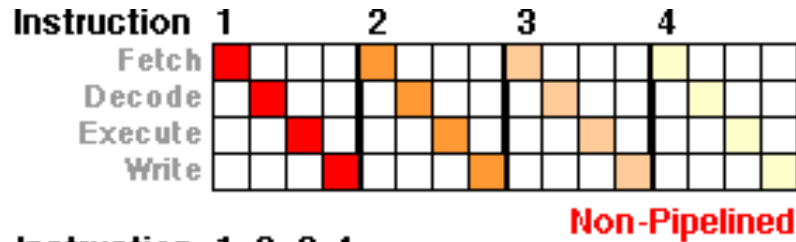
$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6}$$

To improve performance, reduce execution time  $T$   
Reducing CPI improves  $T$



# Simple Pipelining

Simulator: <https://kobzol.github.io/davis/>



Cycles/Time →

- Parts of different instructions execute in parallel in various units
- Pipeline speedup  $\propto$  pipeline stages
- Pipeline stalls reduce performance

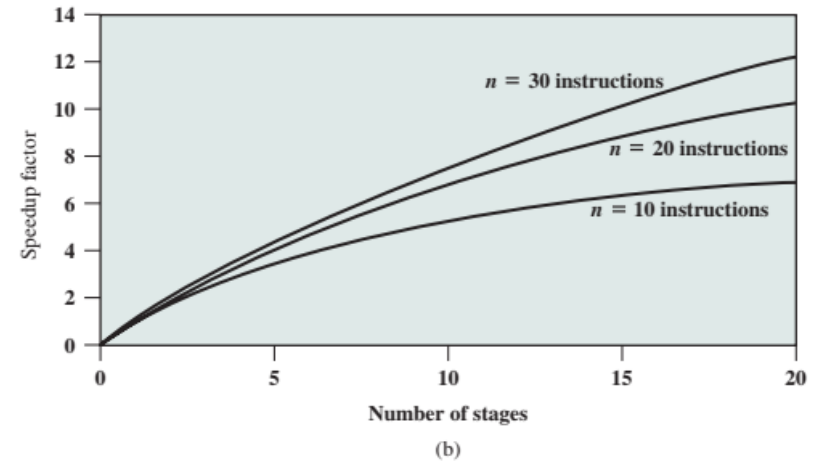


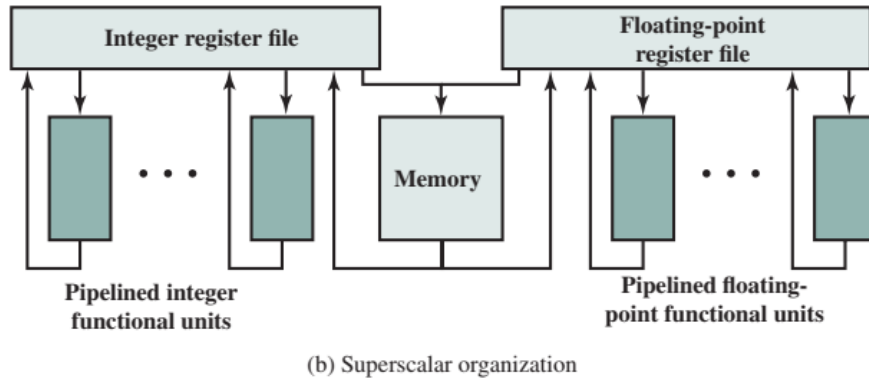
Figure 14.14 Speedup Factors with Instruction Pipelining

$n$  - no. of instructions without branch

# Superpipelined and Superscalar

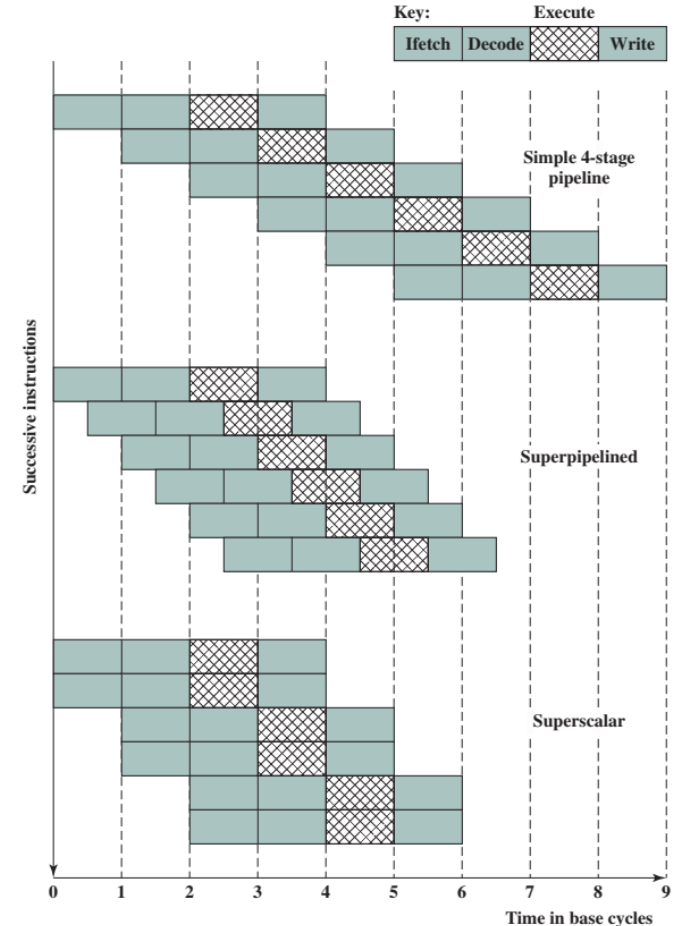
**Superpipelined:** functions performed in each stage can be split into two non overlapping parts and each can execute in half a clock cycle

**Superscalar:** Execution of multiple instructions in parallel by multiple execution units



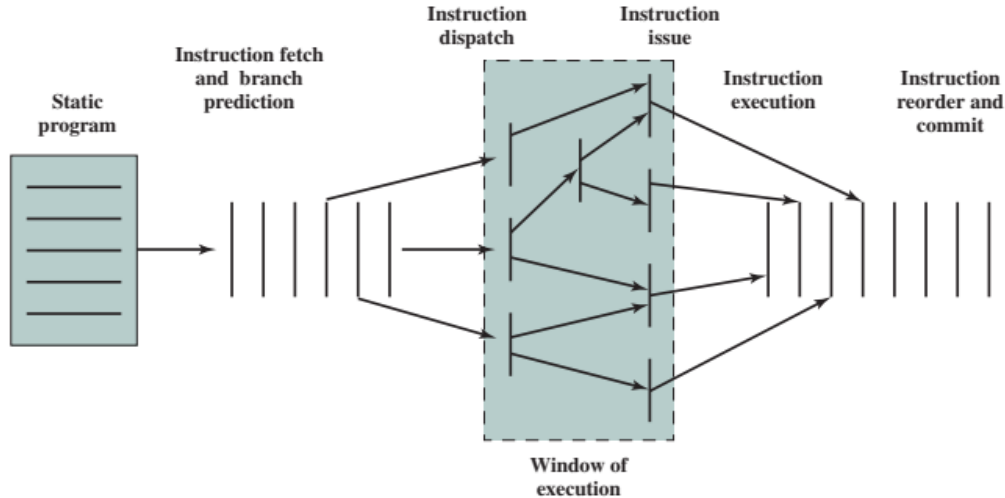
(b) Superscalar organization

**Figure 16.1** Superscalar Organization Compared to Ordinary Scalar Organization



**Figure 16.2** Comparison of Superscalar and Superpipeline Approaches

# Superscalar Execution



**Figure 16.7** Conceptual Depiction of Superscalar Processing

## Key elements needed:

1. Prefetch multiple instructions and branch predict
2. Re-order instructions based on true dependency
3. Parallel issue
4. Parallel execution
5. Commit to maintain program order



# ILP Constraints

- 5 Constraints (*Stallings2016*)
  - True data dependency (RAW)
  - Procedural dependency
  - Resource conflicts
  - Output dependency (WAW)
  - Anti-dependency (WAR)
- Challenges - Dependencies and Hazards (*Hennessy2012*)
  - Data Dependencies
    - True Data Dependency
    - Name Dependency (WAW, WAR)
    - Control Dependency (Procedural dependency)
  - Data Hazards (Name or Control dependence)
    - Read after Write (RAW)
    - Write after Write (WAW)
    - Write after Read (WAR)
- Goal: Preserve strict program order, states, exceptions in parallel execution
- ILP: Fine balance between pipeline starvation vs. stalls due to hazards

# ILP Constraint 1: True Data Dependence (RAW)

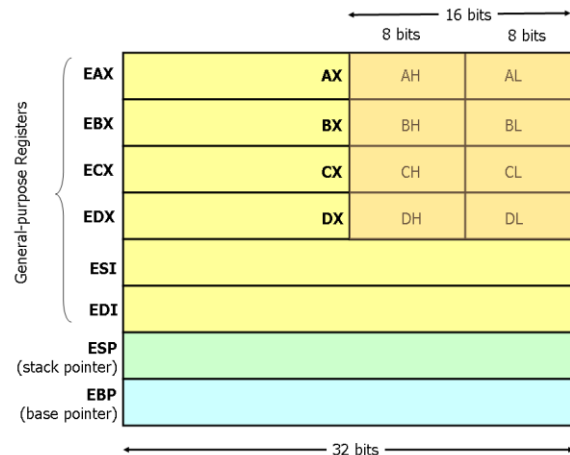
- RAW - Read after Write
  - instruction *i1* updates a register to new value
  - instruction *i2* reads the same register
- Problem:
  - 2nd instruction can't be executed in parallel with 1st (Superscalar)
  - 2nd instruction can't be executed before 1st (OOO)
  - 2nd instruction has to wait till 1st is retired or committed
- Solution:
  - Not really, second instruction has to wait
  - Register renaming won't help

Example 1:

```
ADD EAX, ECX
MOV EBX, EAX
```

Example 2:

```
OR EAX, ECX
... 50 instr later..
MOV EDX, EAX
```

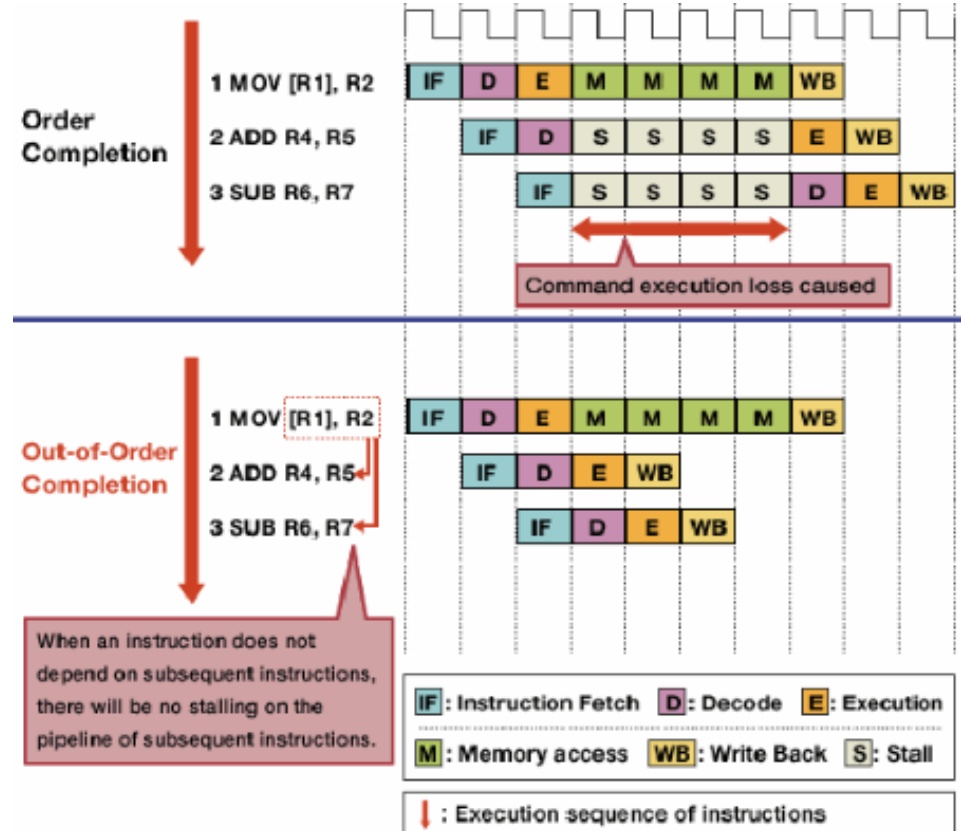


x86 registers

# Out of Order Execution

Superscalar instruction issue policies:

- In-order issue with in-order completion
  - Purely in-order
- In-order issue with out-of-order completion.
  - Issue can be stalled by resource conflict, data or procedural dependency
- Out-of-order issue with out-of-order completion
  - Instructions are pre-fetched (look-ahead)
  - Utilizes **instruction window** to store decoded instructions
  - Decoded instructions scheduled from IW
  - More complicated (during interrupts), but offers best performance



# ILP Constraint 2: Procedural Dependence

- Also called Control dependence
- 2 Types: Uncertainties due to
  - Upcoming branch
  - Variable length instruction fetch (for CISC)
- Problem with superscalar & out-of-order?
  - Branch prediction and
  - Speculative execution helps

Example 1:

```
MOV EAX, 1
CMP EAX, 1
JE .equal
OR EAX, ECX
```

.equal:

```
CALL function
```

Example 2:

```
MOV EAX, [EBX + EDI + 1]
XOR ECX, ECX
```



## ILP Constraint 3: Resource Conflict

- Two or more instructions competing for the same resource (ex. ALU)
- Need additional functional units

Example:

```
MOV EAX, 4  
ADD EBX, 4  
MUL ECX, 5  
SUB EAX, 4
```





## ILP Constraint 4: Output Dependence (WAW)

- Occurs in in-order issue with out-of-order-completion
- Instruction *i3* writes to the same register that instruction *i1* writes
- Solved by **Register Renaming**

**Example:**

i1: **R3** <- R3 op R5

i2: R4 <- R3 + 1

i3: **R3** <- R5 + 1

i4: R7 <- R3 op R4

# ILP Constraint 5: Anti-dependence (WAR)

- Reverse of True dependence
- Instruction *i2* writes to a register that instruction *i1* reads
- *I3* waits for *I2* because *R3* will be updated
- True dependency (RAW):
  - Second instruction uses a value produced by first instruction
- Anti-dependency (WAR):
  - Second instruction destroys a value produced by first instruction
- Solved by **Register Renaming**

## Example:

I1: *R3* <- *R3* op *R5*

I2: *R4* <- ***R3*** + 1

I3: ***R3*** <- *R5* + 1

I4: *R7* <- *R3* op *R4*

# ILP Constraints - Review

1. i2 (read R3) cannot precede by i1 (write R3) due to **True Dependency (RAW)**
2. i5 cannot precede i4 due to branch - **Procedural Dependency**
3. i3 (add) cannot precede i2 (add) due to **Resource Constraint** on adder
4. i3 (write R3) cannot precede i1 (write R3) due to **Output Dependency (WAW)**
5. i3 (write R3) cannot precede i2 (read R3) due to **Anti-dependency (WAR)**

## Example:

```
i1: R3 <- R3 op R5
    i2: R4 <- R3 + 1
    i3: R3 <- R5 + 1
    i4: BR stop if R3 < 0
    i5: R7 <- R3 op R4
stop: i6: R7 <- R3 + 1
```

# Register Renaming

1. Anti-dependencies (WAR) and output dependencies (WAW) are both examples of storage conflicts.
2. **Register renaming** is a technique that abstracts logical registers (ex. EAX, ECX) from physical registers
3. Can eliminate false data dependency between instructions arising due to reuse of registers
4. Processor has set of internal physical registers for each architectural logical registers
5. Internal register handle renaming transparently

Example 1

```
I1: R3 ← R3 op R5
I2: R4 ← R3 + 1
I3: R3 ← R5 + 1
I4: R7 ← R3 op R4
```



```
I1: R3b ← R3a op R5a
I2: R4b ← R3b + 1
I3: R3c ← R5a + 1
I4: R7b ← R3c op R4b
```

Has WAW, i3 can't  
execute OO before i2

After register renaming,  
no WAW

Example 2

```
I1: r1 := m[1024]
I2: r1 := r1 + 2
I3: m[1032] := r1
I4: r1 := m[2048]
I5: r1 := r1 + 4
I6: m[2056] := r1
```



```
r1 := m[1024]
r1 := r1 + 2
m[1032] := r1
r2 := m[2048]
r2 := r2 + 4
m[2056] := r2
```

Has WAW, i4 can't  
execute OO before i3

After register renaming,  
no WAW

# Register Renaming

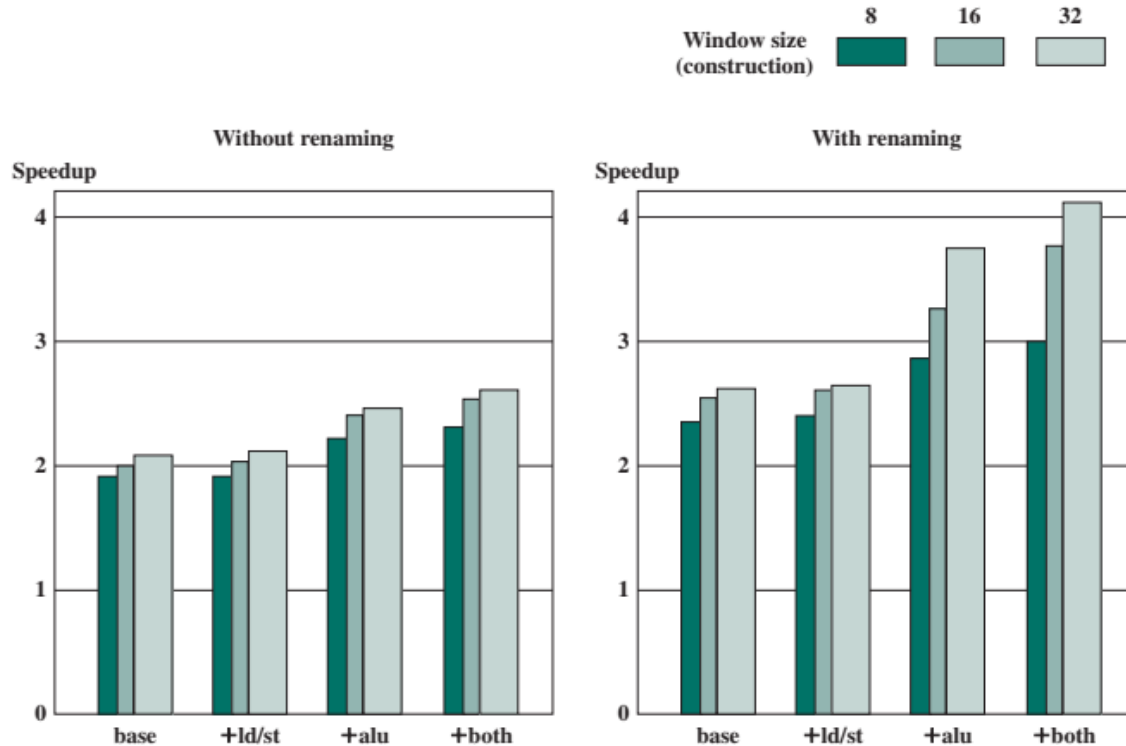


Figure 16.6 Speedups of Various Machine Organizations without Procedural Dependencies

Observations:

- No significant speedup with +ld/st
- Register renaming offers significant speed up with +alu

# Control Dependence (Hennessy2012)

- Control dependence determines the ordering of an instruction,  $i$ , with respect to a branch instruction so that instruction  $i$  is executed in correct program order and only when it should be.
- Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches.
- Two critical properties for program correctness to maintain:
  - Exception behaviours
  - Data flow

```
if p1 {  
    S1;  
};  
if p2 {  
    S2;  
}
```

# Control Dependence - Preserve Exception Behavior

- Data dependence between DADDU and BEQZ because of R2 (RAW)
- Between BEQZ and LW, no dependence but control dependence
- Swapping BEQZ and LW can cause exception
- Need to preserve exceptions behaviour
- **Speculative Execution** allows executing LW before BEQZ

	DADDU	R2, R3, R4
	BEQZ	R2, L1
	LW	R1, 0(R2)
L1:		

# Control Dependence - Preserve Data flow

- The data flow is the actual flow of data values among instructions that produce results and those that consume them.
- Branches make the data flow dynamic
- In example below:
  - Final R1 value depends on branch
  - DSUBU can move before BEQZ?
  - Yes, with speculative execution

```
L:      DADDU      R1,R2,R3
        BEQZ      R4,L
        DSUBU     R1,R5,R6
        ...
        OR        R7,R1,R8
```

## *Liveness* detection

- In example below, if R4 unused after skip, moving DSUBU before BEQZ won't cause problems since it's dead. Data flow not affected
- Else, R4 will be live. OOE of DSUBU can cause exception
- Example of software speculation

```
        DADDU     R1,R2,R3
        BEQZ     R12,skip
        DSUBU    R4,R5,R6
        DADDU    R5,R4,R9
skip:    OR       R7,R8,R9
```





# Compiler Techniques for ILP - Loop unrolling

- Goal is to reduce pipeline stalls due to branches
- The execution of a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction
- Compiler can enhance ILP by
  - Identifying parallelism opportunities (ex. Loop unrolling)
  - Pipeline scheduling
  - Hints for branch prediction

- Compiler has view of the entire program
- Has the luxury of time (build vs run time)

Loop unrolling - a compiler technique, not hardware technique

## Loop unrolling

- Unrolling simply replicates the loop body multiple times, adjusting the loop termination code
- Problem 1: Sometimes we do not know the upper bound of the loop
- Problem 2: Unrolling a large loop increases program size, and may erase any performance gains



# Parallel and Dependent loops

```
for(i=999; i>=0; i=i-1)
```

```
  x[i] = x[i] + s;
```

- Independent loops are parallelizable
- Each iteration does not depend on data from previous or next iteration

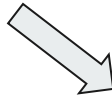
```
for(i=999; i>0; i=i-1)
```

```
  x[i] = x[i-1] + s;
```

- This example shows dependency on next iteration
- Cannot be parallelized, so loop unrolling cannot be done

# Compiler - Unroll Parallel loops

```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```



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

Initialization:

R1 - address of highest array  
element

F2 - contains value of s

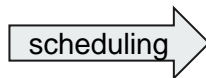
R2 - address of the last array  
element

Straightforward translation from C to MIPS assembly

# Scheduling to optimize

## Clock cycle issued

Loop:	L.D	F0,0(R1)	1
	<i>stall</i>		2
	ADD.D	F4,F0,F2	3
	<i>stall</i>		4
	<i>stall</i>		5
	S.D	F4,0(R1)	6
	DADDUI	R1,R1,#-8	7
	<i>stall</i>		8
	BNE	R1,R2,Loop	9



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

MIPS code, not scheduled for the pipeline.  
Takes 9 clock cycles per element

After scheduling, takes 7 cycles per element

3 for actual work, 4 for loop overhead + stalls

# Loop Unrolling + Scheduling Optimization

```
for(i=999; i>=0; i = i-4) {  
    x[i] = x[i] + s;  
    x[i+1] = x[i+1] + s;  
    x[i+2] = x[i+2] + s;  
    x[i+3] = x[i+3] + s;  
}
```

For representation only. Compiler will do loop unrolling, programmer need not!

Loop:

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

scheduling

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)
S.D	F16,8(R1)
BNE	R1,R2,Loop

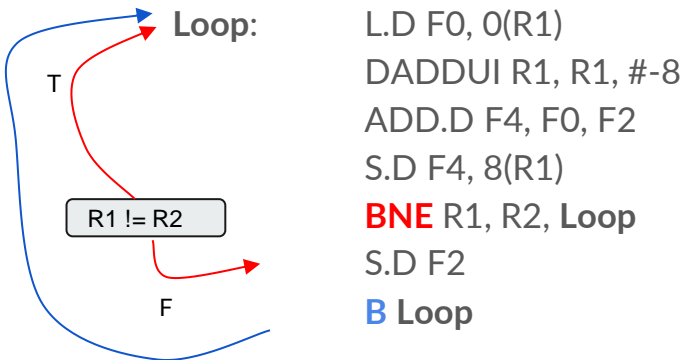
After loop unrolling, it takes 27 clock cycles  
(6.75 clocks per element)

After scheduling, it takes 14 clock cycles  
(3.5 clocks per element)

# Branch Prediction

*What is a branch?*

**Conditional** or **Unconditional** change in execution of instruction sequence



*What is branch prediction?*

Determining a branch's direction and target before its execution, i.e immediately after fetch

*Why branch predict?*

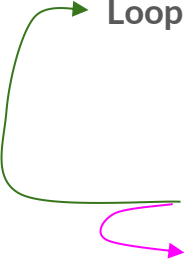
- In a superscalar pipeline, the processor fetches the next instruction even before completing the previous instructions by incrementing program counter.
- This won't work for branches, since it has to fetch from the branch target address.
- Several cycles will be lost (CPI increases) and pipeline may stall if processor waits for branch instruction to be executed.
- It's not possible to predict with 100% accuracy.
- Need to predict branch target at > 80% accuracy in 1-2 cycles

# Branch Prediction

Few techniques:

- Static approaches
  - Predict never taken
  - Predict always taken
  - Predict by opcode (~70%)
- Dynamic (history based)
  - Taken/not taken switch
  - Branch history table

Loop:



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

## Taken/not taken switch

- Maintain history bit/bits per branch
- 1-bit or 2-bit
- Drawback: Need to decode branch instruction to get the target

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

# Branch History Table

- BHT is updated with prediction initially
- BHT is updated with actual direction after branch
- Modern high-performance processors have branch misprediction delays on the order of 15 clock cycles; clearly, accurate prediction is critical!

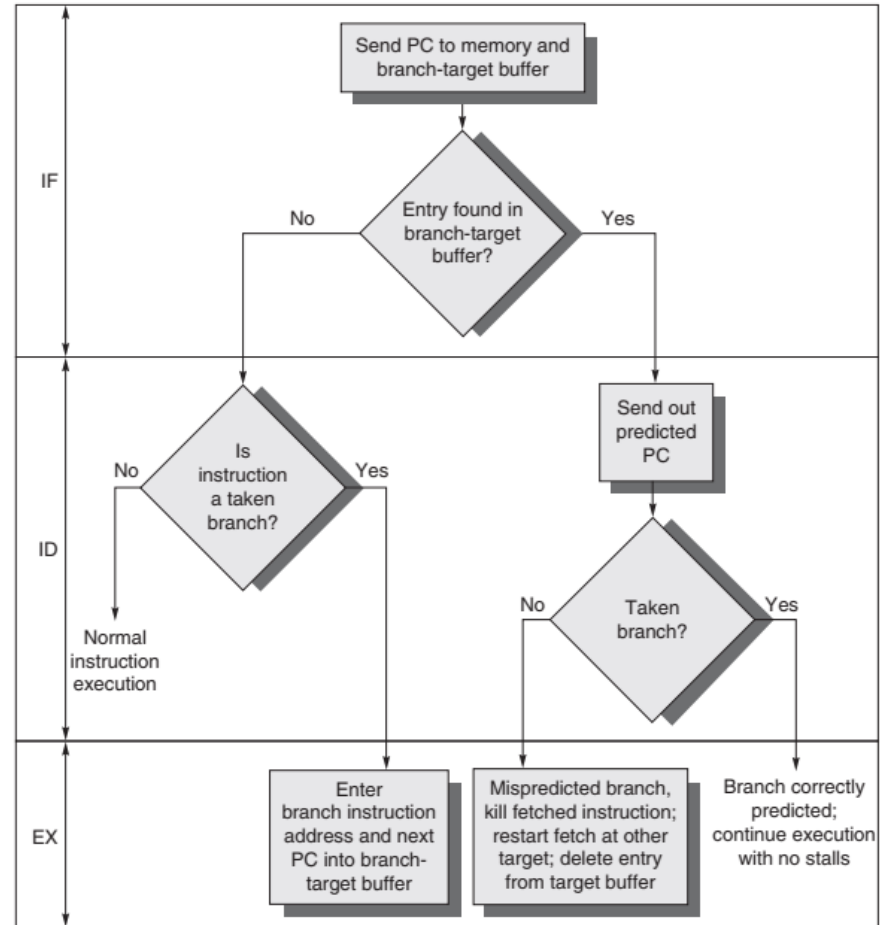


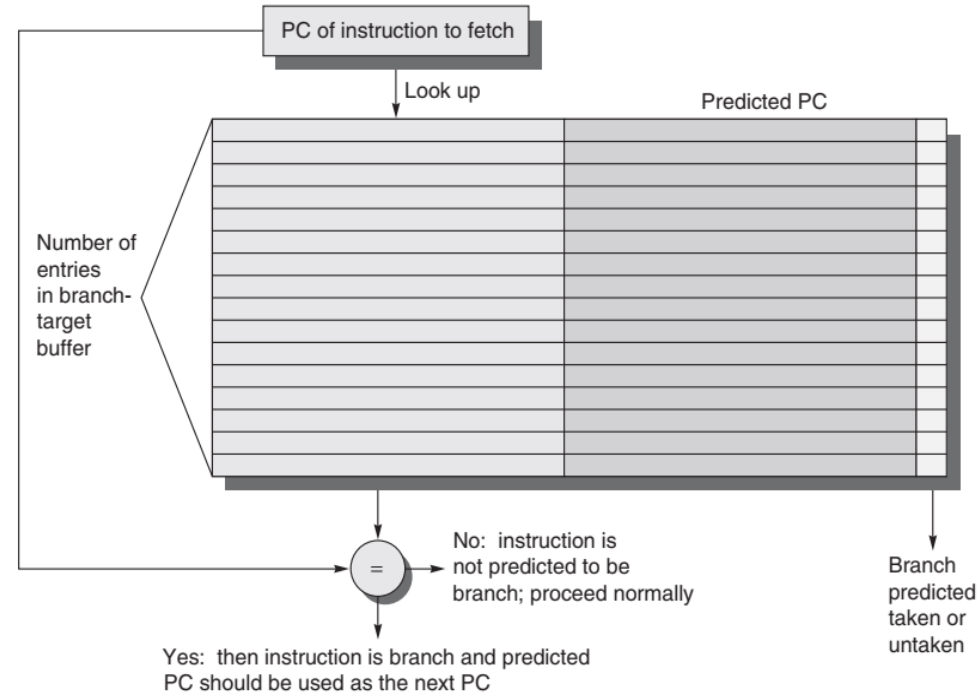
Figure 3.22 The steps involved in handling an instruction with a branch-target buffer.



# Branch History Table

Branch history table (Branch target buffer)

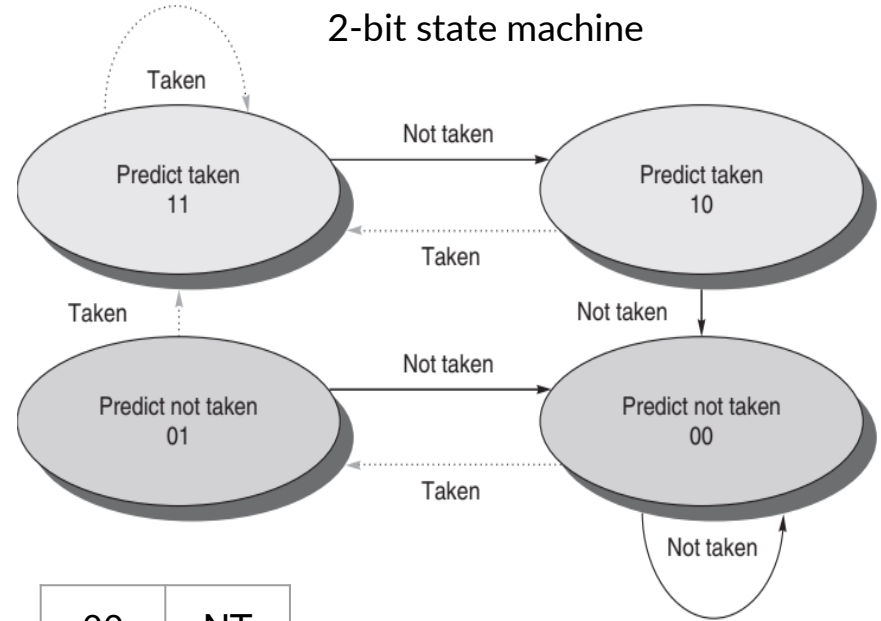
- Every instruction is checked in BHT at fetch
- When branch is taken, target address is entered into BHT
- If out of space, flush an older entry
- During execution of branch instruction, if branch misses, then entry updated into BHT and redirect to correct address



## 2-bit scheme

- Possible values = 4 (00, 01, 10, 11)
  - Initialize value to 11
  - If value  $\geq 2$ , predict taken, else not taken
  - If branch is actually taken, shift 1, else 0
- 
- General n-bit scheme
  - Total possible values =  $2^n$
  - Values from 0 to  $2^n - 1$

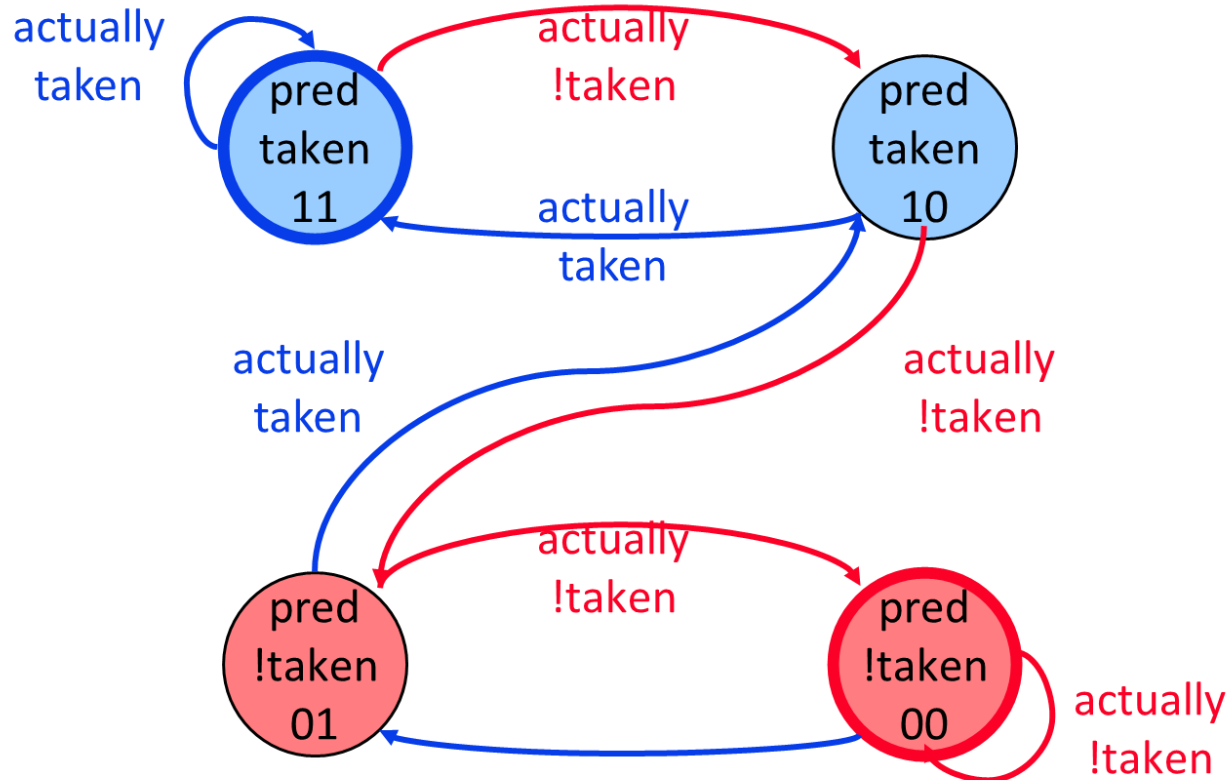
With 4096 entries, accuracy improved between 82% to 99%



00	NT
01	NT
10	T
11	T

2-bit state table

## 2-bit saturating counter (State machine)



# State transitions

Predicted state	Actual state	History bits (1-bit)
T		1
T	T	1
T	T	1
T	NT	0
NT	NT	0
NT	T	1
T	NT	0
NT	T	1

Predicted state	Actual state	History bits (2-bit)
T		1 1
T	T	1 1
T	T	1 1
T	NT	1 0
T	NT	0 0
NT	T	0 1
NT	T	1 1
T	NT	1 0

# Branch History Table (initial state)

Address	Instruction	Instruction Address	Branch Target	History bits
100	Loop: L.D F0, 0(R1)			
104	DADDUI R1, R1, #-8			
108	ADD.D F4, F0, F2			
112	S.D F4, 8(R1)			
118	BNE R1, R2, Loop			
122	S.D F2			
128	B Loop			

If initial state = 1 1 or 1 0, T T T T T T T T T T N

If initial state = 0 0 or 0 1, T T T T T T T T T T N

## Branch History Table (After 1st branch)

Address	Instruction	Instruction Address	Branch Target	History bits
100	Loop: L.D F0, 0(R1)			
104	DADDUI R1, R1, #-8			
108	ADD.D F4, F0, F2			
112	S.D F4, 8(R1)			
118	<b>BNE R1, R2, Loop</b>	118	100	1 1
122	S.D F2			
128	B Loop			

## Branch History Table (loop exit)

Address	Instruction	Instruction Address	Branch Target	History bits
100	Loop: L.D F0, 0(R1)			
104	DADDUI R1, R1, #-8			
108	ADD.D F4, F0, F2			
112	S.D F4, 8(R1)			
118	<b>BNE R1, R2, Loop</b>	118	100	1 0
122	S.D F2			
128	B Loop			

# Branch Prediction - Advanced

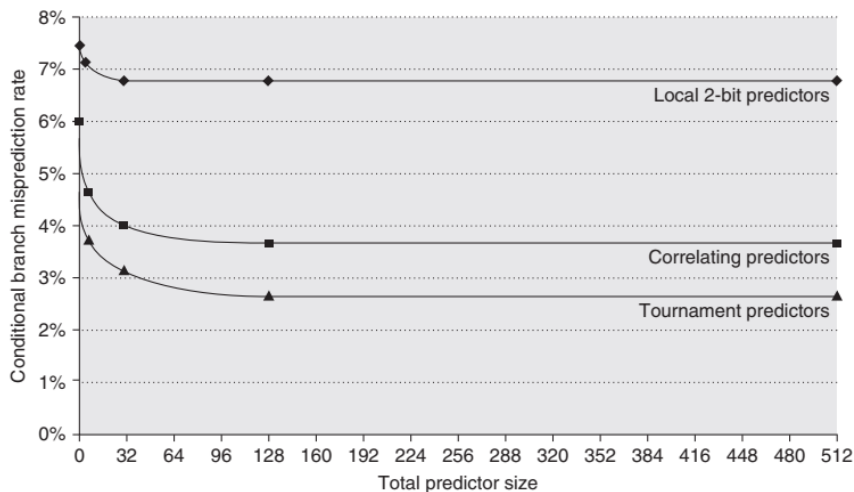
## Correlating Branch Predictors

- Sometimes branches are dependent on previous branches
- Correlating BP uses history of  $m$  previous branches, each branch having  $n$  bit states
- $(m,n)$  predictor
- Table space reqd (in bits) =
  - $2^m * n$  entries

```
if (aa==2)
    aa=0;
if (bb==2)
    bb=0;
if (aa!=bb) {
```

## Tournament Branch Predictors

- Uses multiple predictors
- Adaptively combines global and local BP
- Has ability to select the right predictor
- Performs well for integer benchmarks







# Speculative Execution

Combines three key ideas:

- **Dynamic branch prediction** to choose which instructions to execute
- **Speculation** to allow the execution of instructions before the control dependencies are resolved (with the ability to undo the effects of an incorrectly speculated sequence), and
- **Dynamic scheduling** to deal with the scheduling of different combinations of basic blocks

Typically accomplished using Reorder Buffer (ROB)

- ROB holds the result of an instruction between the time the operation associated with the instruction completes and the time the instruction commits
- Used to pass results among instructions that may be speculated
- Contains four fields: the instruction type, the destination field, the value field, and the ready field

# Dynamic Scheduling

## Tomasulo's Algorithm

- Originally developed for IBM 360/91 floating point unit

3 Key concepts implemented:

- Register renaming
- Reservation stations: Handles WAW and WAR hazards by register renaming
- Common Data Bus: broadcasts results to waiting instructions in reservation stations

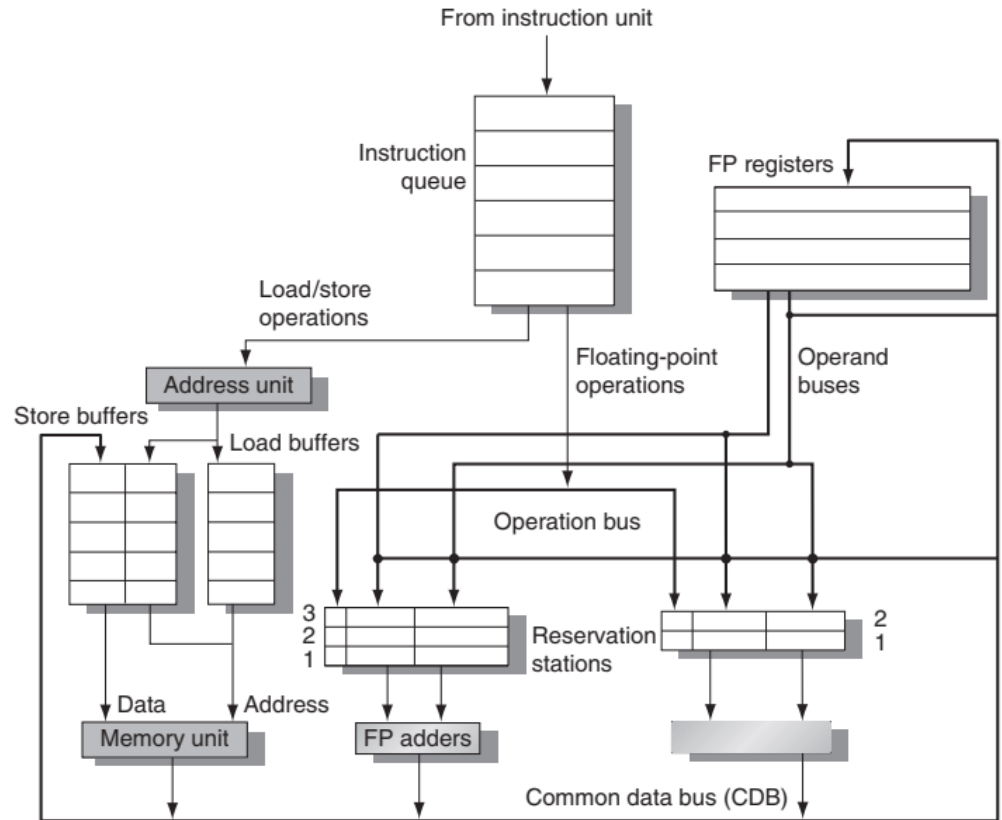


Figure 3.6 The basic structure of a MIPS floating-point unit using Tomasulo's algorithm

# Tomasulo's Algorithm - Example

Instruction status

Instruction	Issue	Execute	Write result
L.D F6, 32(R2)	✓	✓	✓
L.D F2, 44(R3)	✓	✓	
MUL.D F0, F2, F4	✓		
SUB.D F8, F2, F6	✓		
DIV.D F10, F0, F6	✓		
ADD.D F6, F8, F2	✓		

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs[R3]
Add1	Yes	SUB		Mem[32 + Regs[R2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs[F4]	Load2		
Mult2	Yes	DIV		Mem[32 + Regs[R2]]	Mult1		

Register status

Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

- Op—The operation to perform on source operands S1 and S2.
- Qj, Qk—The reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary.
- Vj, Vk—The value of the source operands. Note that only one of the V fields or the Q field is valid for each operand. For loads, the Vk field is used to hold the offset field.
- A—Used to hold information for the memory address calculation for a load or store. Initially, the immediate field of the instruction is stored here; after the address calculation, the effective address is stored here.
- Busy—Indicates that this reservation station and its accompanying functional unit are occupied.



# ILP - design questions

## Design questions

1. Calculate MIPS given clock, miss rates, CPI
2. Calculate speedup with superscalar pipelining and dynamic scheduling
3. Identify various dependencies and hazards. Suggest solutions to resolve hazards
4. Calculate speedup by loop unrolling and scheduling
5. Compare various branch prediction schemes for given code.
6. Compute branch prediction accuracy for a given dynamic or static scheme
7. Compute prediction states of a 2-bit branch predictor given a branch sequence and initial state

# KC1 - ASM Basics

Assume Memory[100] = 10

No	Instruction	R0	R1	R2	R3	R4	R5	R6	R7
	<i>Initial values -&gt;</i>	8	25	75	10	1	7	20	34
0	<b>ADD</b> R3, R1, R2				100				
1	<b>LOAD</b> R6, [R3]							10	
2	<b>AND</b> R7, R5, 3				3				
3	<b>ADD</b> R1, R6, R7		44						
4	<b>SRL</b> R7, R0, 1	4							
5	<b>OR</b> R2, R4, R7								
6	<b>SUB</b> R5, R3, R4								
7	<b>ADD</b> R0, R1, 10								
	<i>Compute final values</i>	?	?	?	?	?	?	?	?

# KC2 - Pipelining

## Assumptions:

- 4 stage pipeline: Fetch (F), Decode (D), Execute (E), Write back (W)
- F, D, W = 1 cycle each
- E: LOAD = 5, others = 1 cycle

	Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	<b>ADD</b> R3, R1, R2	F	D	E	W														
1	<b>LOAD</b> R6, [R3]		F	D		E					W								
2	<b>AND</b> R7, R5, 3			F	D		E	W											
3	<b>ADD</b> R1, R6, R7				F	D						E	W						
4	<b>SRL</b> R7, R0, 1					F	D	E	W										
5	<b>OR</b> R2, R4, R7						F	D		E	W								
6	<b>SUB</b> R5, R3, R4							F	D		E	W							
7	<b>ADD</b> R0, R1, 10								F	D				E	W				

Assume: **Scalar, Out of Order** execution

Stallings2016, page 609, question 16.2 a)

# KC2 - Pipelining

## Assumptions:

- 4 stage pipeline: Fetch (F), Decode (D), Execute (E), Write back (W)
- F, D, W = 1 cycle each
- E: LOAD = 5, others = 1 cycle

	Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	<b>ADD</b> R3, R1, R2	F	D	E	W														
1	<b>LOAD</b> R6, [R3]		F	D		E					W								
2	<b>AND</b> R7, R5, 3			F	D						E	W							
3	<b>ADD</b> R1, R6, R7				F	D							E	W					
4	<b>SRL</b> R7, R0, 1					F	D							E	W				
5	<b>OR</b> R2, R4, R7						F	D								E	W		
6	<b>SUB</b> R5, R3, R4							F	D								E	W	
7	<b>ADD</b> R0, R1, 10								F	D								E	W

Assume: **Scalar, In-Order execution**

Stallings2016, page 609, question 16.2 b)

# KC2 - Pipelining

## Assumptions:

- 4 stage pipeline: Fetch (F), Decode (D), Execute (E), Write back (W)
- F, D, W = 1 cycle each
- E: LOAD = 5, others = 1 cycle

	Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	<b>ADD</b> R3, R1, R2	F	D	E	W														
1	<b>LOAD</b> R6, [R3]	F	D			E					W								
2	<b>AND</b> R7, R5, 3		F	D	E	W													
3	<b>ADD</b> R1, R6, R7		F	D								E	W						
4	<b>SRL</b> R7, R0, 1			F	D	E	W												
5	<b>OR</b> R2, R4, R7			F	D			E	W										
6	<b>SUB</b> R5, R3, R4				F	D	E	W											
7	<b>ADD</b> R0, R1, 10				F	D								E	W				

Assume: 2-issue superscalar, Out-of-Order execution

Stallings2016, page 609, question 16.2 c)





## KC2 - Summary

- Scalar, out-of-order = X cycles
- Scalar, in-order = Y cycles
- 2-issue superscalar, OOO = Z cycles

Exercise: Complete the tables for all 10 instructions and find values of X, Y, Z

## KC3 - Hazard Identification - 1

No	Instruction	Hazard	Reason
I1	<b>Move</b> R3, R7	None	
I2	<b>Load</b> R8, (R3)	R3:RAW	I2 reads R3, which I1 writes
I3	<b>Add</b> R3, R3, 4		
I4	<b>Load</b> R9, (R3)		
I5	<b>BLE</b> R8, R9, L3		

Exercise: Complete the table

*Stallings2016*, page 609, question 16.3

## KC3 - Hazard Identification - 2

No	Instruction	Hazard	Solution - Register Renaming
I1	<b>R1 = 100</b>	None	R1a = 100
I2	<b>R1 = R2 + R4</b>	R1:WAW	R1b = R2 + R4
I3	<b>R2 = R4 - 25</b>		
I4	<b>R4 = R1 + R3</b>		
I5	<b>R1 = R1 + 30</b>		

Exercise: Complete the table

*Stallings2016*, page 609, question 16.4



## KC4 - Branch Prediction - 1 bit history

Calculate branch misprediction rate and CPI for 1-bit and 2-bit schemes

Given:

- BHT hit accuracy (A)
- Base CPI (B)
- Branch misprediction delay in cycles (C)
- Actual branch sequence
- Initial state state

Steps:

1. Calculate the total branches and no. of mispredictions, and misprediction rate (M)
2. Calculate branch misprediction rate
3. Calculate CPI using formula

$$\text{CPI} = B + [(1 - A) * (M) * C]$$

# KC4 - Branch Prediction - 1 bit history

Calculate branch misprediction rate and CPI  
Assume the following values

Attribute	Value
BHT hit accuracy (A)	80%
Base CPI (B)	1
Misprediction delay (C)	2
Initial state	1

Given actual branch sequence:  
TTTTNNNN

State bit	Prediction	Actual
1	T	T
1	T	T
1	T	T
1	T	T
1	T	N
0	N	N
0	N	N
0	N	N

## KC4 - Branch Prediction - 1 bit history

Successful predictions = 7

Mispredictions = 1

Misprediction rate (M) = (mispredictions/total) =  $1/(1+7) = 0.125$

$$\begin{aligned}\text{CPI} &= B + [(1 - A) * (M) * C] \\ &= 1 + [(1 - 0.8) * 0.125 * 2] = 1.05\end{aligned}$$

CPI increases by 0.05 for 1-bit scheme

**Exercise (with same assumptions):**

1. Calculate CPI for: TNTNTNTNTNTN
2. Calculate CPI for: TTNNTTNNTTNN

# KC4 - Branch Prediction - 2 bit history

Calculate branch misprediction rate and CPI  
Assume the following values

Attribute	Value
BHT hit accuracy (A)	90%
Base CPI (B)	1
Misprediction delay (C)	3
Initial state	1 0

Given actual branch sequence:  
TTTTNNNN

State bits	Prediction	Actual
1 0	T	T
1 1	T	T
1 1	T	T
1 1	T	T
1 1	T	N
1 0	T	N
0 0	N	N
0 0	N	N

## KC4 - Branch Prediction - 2 bit history

Successful predictions = 6

Mispredictions = 2

Misprediction rate (M) = (mispredictions/total) =  $2/(2+6) = 0.25$

$$\begin{aligned}\text{CPI} &= B + [(1 - A) * (M) * C] \\ &= 1 + [(1 - 0.9) * 0.25 * 3] = 1.075\end{aligned}$$

CPI increases by 0.075 for 2-bit scheme

**Exercise (with same assumptions) for 2-bit history:**

1. Calculate CPI for: TNTNTNTNTNTN
2. Calculate CPI for: TTNNTTNNTTNN



---

# Data Level Parallelism



# Data Level Parallelism

- What is DLP?
- Parallel Data
- SIMD vs MIMD
- SIMD variations
  - Vector Architecture (SIMD)
  - Multimedia SIMD instructions
  - GPU

# Basics - Quiz

1. Byte, Word, Double word, Quad Word
  2. Single vs Double Precision?
  3. Scalar/Vector? (Maths, Physics, C++, DLP)
- Byte (8), Word (16), Double word (32), Quad word (64)
  - Single Precision: 32-bit, Double precision: 64-bit, Quadruple precision: 128-bit

	Mathematics	Physics	C++	DLP
Scalar	Real numbers	Quantities that are fully described by a magnitude (or numerical value) alone (e.g mass, distance, etc)	Integers, float, char, etc	Integers, float, char, etc
Vector	Has magnitude and direction. Dot product, cross product, vector algebra	Quantities that are fully described by both a magnitude and a direction (e.g velocity, force, etc)	Dynamic array - ADT  Ex: <code>vector&lt;int&gt; names (n, 1);</code>	1-d array of fixed size

Example

$$\begin{bmatrix} 2 \\ 5 \\ 1 \\ 8 \end{bmatrix}$$



# Why SIMD ?

- Scientific Computing, Multimedia – all examples of vector data, where operations are mostly parallel
- ILP techniques can help only to an extend

Example of Vector Operation

$$Y = a \times X + Y$$

(SAXPY stands for single-precision  $a \times X$  plus  $Y$ ; DAXPY for double precision  $a \times X$  plus  $Y$ )

- MMX, SSE, AVX – examples of SIMD instructions in x86 architecture

# VMIPS instructions

Instruction	Operands	Function
ADDVV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBVV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULVV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2 (i.e., $R1 + i \times R2$ ).
SVWS	(R1,R2),V1	Store V1 to address at R1 with stride in R2 (i.e., $R1 + i \times R2$ ).
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at $R1 + V2(i)$ (i.e., V2 is an index).
SVI	(R1+V2),V1	Store V1 to vector whose elements are at $R1 + V2(i)$ (i.e., V2 is an index).

# Vector Registers



V1



V2

## Option 1: Scalar method

```
for (int i = 0; i < 63; i++) {  
    V3[i] = V1[i] + V2[i];  
}
```

## Option 2: Vector method

ADDVV.D V3, V1, V2



# MIPS: C to Assembly

- Compiler Explorer
  - Link: [Online Assembly](#)
  - Language: C++
  - Compiler: mips gcc 12.2.0
  - Compiler Options: -std=gnu++14 -Wall -Wextra -fno-verbose-asm -ffast-math
- Command line (gcc) for MIPS on Linux:
  - `$ sudo apt-get install -y gcc-mips-linux-gnu` (Installation of cross compiler)
  - `$ mips-linux-gnu-gcc -g -c -march=mips32 vector.c`
  - `$ mips-linux-gnu-objdump -M mips64 -d -S vector.o > vector.s`
  - `$ vi vector.s` (*assembly file*)

# SIMD Comparison

	L.D	F0,a	;load scalar a
	DADDIU	R4,Rx,#512	;last address to load
Loop:	L.D	F2,0(Rx)	;load X[i]
	MUL.D	F2,F2,F0	;a × X[i]
	L.D	F4,0(Ry)	;load Y[i]
	ADD.D	F4,F4,F2	;a × X[i] + Y[i]
	S.D	F4,9(Ry)	;store into Y[i]
	DADDIU	Rx,Rx,#8	;increment index to X
	DADDIU	Ry,Ry,#8	;increment index to Y
	DSUBU	R20,R4,Rx	;compute bound
	BNEZ	R20,Loop	;check if done

Vector sequence: 6 instrs

Scalar Loop: ~600 instrs.

L.D	F0,a	;load scalar a
LV	V1,Rx	;load vector X
MULVS.D	V2,V1,F0	;vector-scalar multiply
LV	V3,Ry	;load vector Y
ADDVV.D	V4,V2,V3	;add
SV	V4,Ry	;store the result

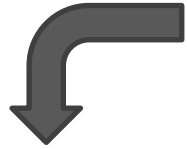




# Vector Execution

- Execution time of Vector instructions is variable, depends on:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- For simplicity: in VMIPS, assume execution time – proportional to vector length
- **Convoy** – chain of related vector instructions
- **Chimes** – time taken to execute 1 convoy

# Convoy and Chime



LV  
MULVS.D  
LV  
ADDVV.D  
SV

V1,Rx  
V2,V1,F0  
V3,Ry  
V4,V2,V3  
V4,Ry

;load vector X  
;vector-scalar multiply  
;load vector Y  
;add two vectors  
;store the sum

1.	LV	MULVS.D
2.	LV	ADDVV.D
3.	SV	

Convoy 1

Convoy 2

Convoy 3

Assume vector execution is proportional to vector size

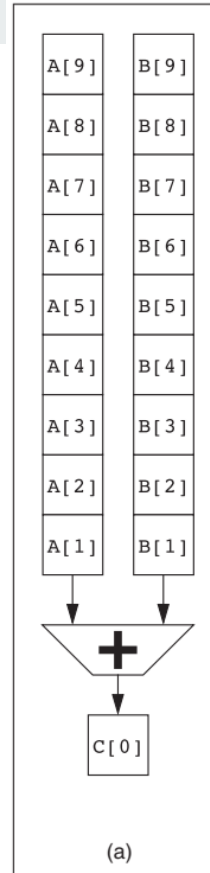
i.e., for vector size 64, clocks taken = 64

No. of chimes = 3

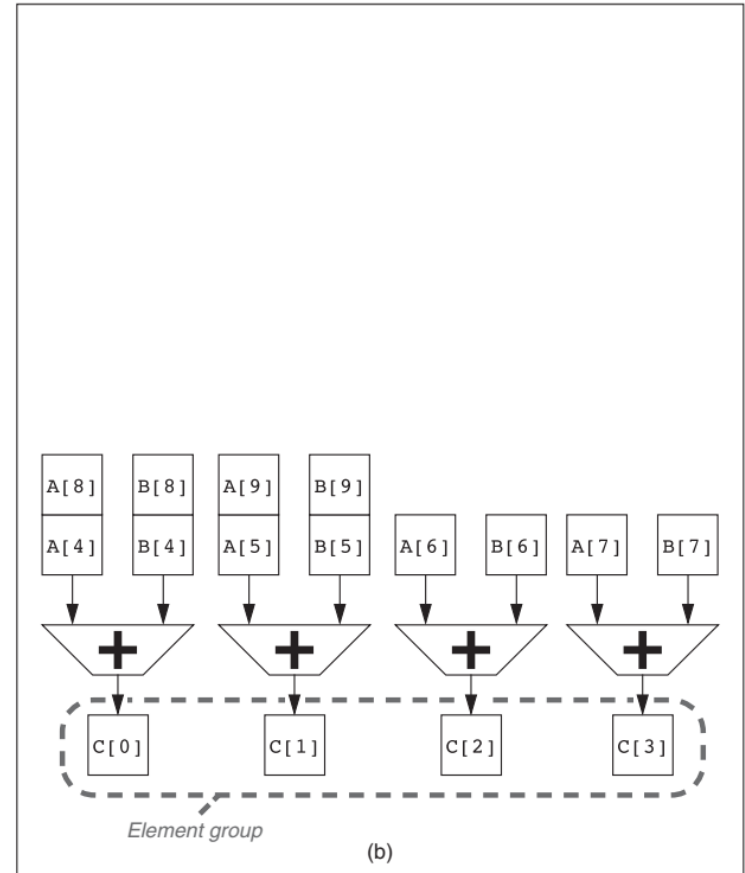
Total clocks (approx.) =  $64 \times 3 = 192$

# Optimization 1: Multiple Lanes

- How can a vector processor execute a single vector faster than one element per clock cycle?
- **Lane:** Parallel execution pipeline with 1 functional unit
- Doubling lanes doubles performance



Single lane  
1 Chime = 64 clocks



4 lanes  
1 Chime = 16 clocks

## Optimization 2: Variable length vectors

- How does a vector processor handle programs where the vector lengths are not the same as the length of the vector register (64 for VMIPS)?
- **Strip Mining** technique
- Handles variable size vectors
- Set the **VLR (Vector Length Register)** register accordingly

```
for (i=0; i <n; i=i+1)
    Y[i] = a * X[i] + Y[i];
```

*Variable size vector (C example)*

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
    for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
        Y[i] = a * X[i] + Y[i] ; /*main operation*/
    low = low + VL; /*start of next vector*/
    VL = MVL; /*reset the length to maximum vector length*/
}
```

*Strip mining – C implementation*

*Instructions to set/read VLR register*

MTC1  
MFC1

VLR,R1  
R1,VLR

Move contents of R1 to vector-length register VL.  
Move the contents of vector-length register VL to R1.

# Optimization 3: Vector Mask Register

- What happens when there is an IF statement inside the code to be vectorized?
- Setup mask register VMR
- VMR => 64 bits wide
- If bit == 1, process, else don't

```
for (i = 0; i < 64; i=i+1)
    if (X[i] != 0)
        X[i] = X[i] - Y[i];
```

LV	V1,Rx	;load vector X into V1
LV	V2,Ry	;load vector Y
L.D	F0,#0	;load FP zero into F0
SNEVS.D	V1,F0	;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D	V1,V1,V2	;subtract under vector mask
SV	V1,Rx	;store the result in X

0	9	0	0	0	4	0	0
0	0	6	0	0	0	1	0
0	0	0	5	0	0	1	0
0	0	0	0	0	0	3	0
0	0	6	0	0	0	0	0

*Sparse Matrix*

0	1	0
0	0	1
0	0	0
0	0	0
0	0	1

VMRs

# Optimization 4: Stride for multi-dimensional vectors

- Consider **Matrix multiplication**
- Vector [a1, a2, a3] is sequentially stored in memory
- But vector [b1, b4, b7] is not sequential
- How to fetch this vector? Striding
- This distance separating elements to be gathered into a single register is called the **stride**

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

a1	a2	a3	a4	a5	a6	a7	a8	a9
----	----	----	----	----	----	----	----	----

b1	b2	b3	b4	b5	b6	b7	b8	b9
----	----	----	----	----	----	----	----	----

Row-major organization in Memory

```

for (i = 0; i < 100; i=i+1)
  for (j = 0; j < 100; j=j+1) {
    C[i][j] = 0.0;
    for (k = 0; k < 100; k=k+1)
      C[i][j] = C[i][j] + A[i][k] *
        B[k][j];
  }

```

Instructions that support stride

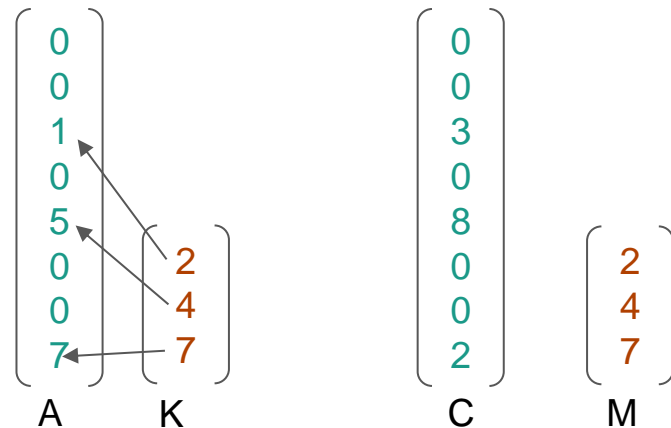
LVWS	V1, (R1, R2)	Load V1 from address at R1 with stride in R2 (i.e., $R1 + i \times R2$ ).
SVWS	(R1, R2), V1	Store V1 to address at R1 with stride in R2 (i.e., $R1 + i \times R2$ ).

# Optimization 5: Gather-Scatter for Sparse Matrix

- Sparse matrices are usually compacted to save space
- VMIPS instruction can take in index vectors and perform Gather-scatter
- Avoids unnecessary addition of 0's

```

LV      Vk, Rk      ;load K
LVI     Va, (Ra+Vk) ;load A[K[]]
LV      Vm, Rm      ;load M
LVI     Vc, (Rc+Vm) ;load C[M[]]
ADDVV.D Va, Va, Vc  ;add them
SVI     (Ra+Vk), Va ;store A[K[]]
    
```



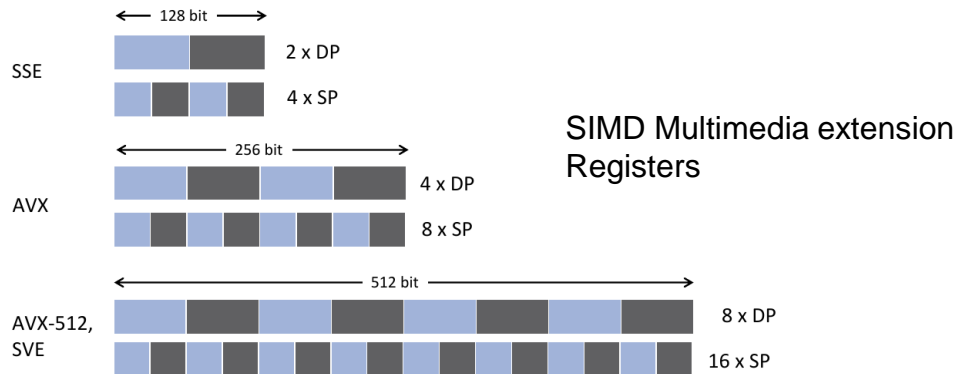
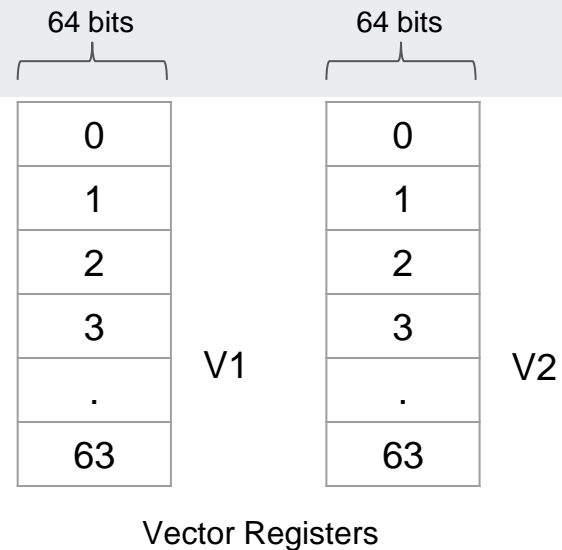
```

for (i = 0; i < n; i=i+1)
    A[K[i]] = A[K[i]] + C[M[i]];
    
```

LVI	V1, (R1+V2)	Load V1 with vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
SVI	(R1+V2), V1	Store V1 to vector whose elements are at R1 + V2(i) (i.e., V2 is an index).

# SIMD Multimedia Extensions

- Multimedia (audio, video, image) data consist of data which is < 32bit
- Graphics 8-bit each for RGB, 8-bit for transparency, Audio sample – 8 or 16-bits
- Vector – Lot of registers! (64 x 64-bit)
  - Less loops. More operations handled in single instruction
  - Complex to implement
  - Operand size is variable (VRL register)
- Multimedia SIMD – smaller set of registers (partition wide registers into smaller parts)
  - More loops – but much less iterations
  - Easier to implement
  - Operand size is fixed
- MMX, SSE, AVX – Multimedia SIMD extensions in x86 architecture



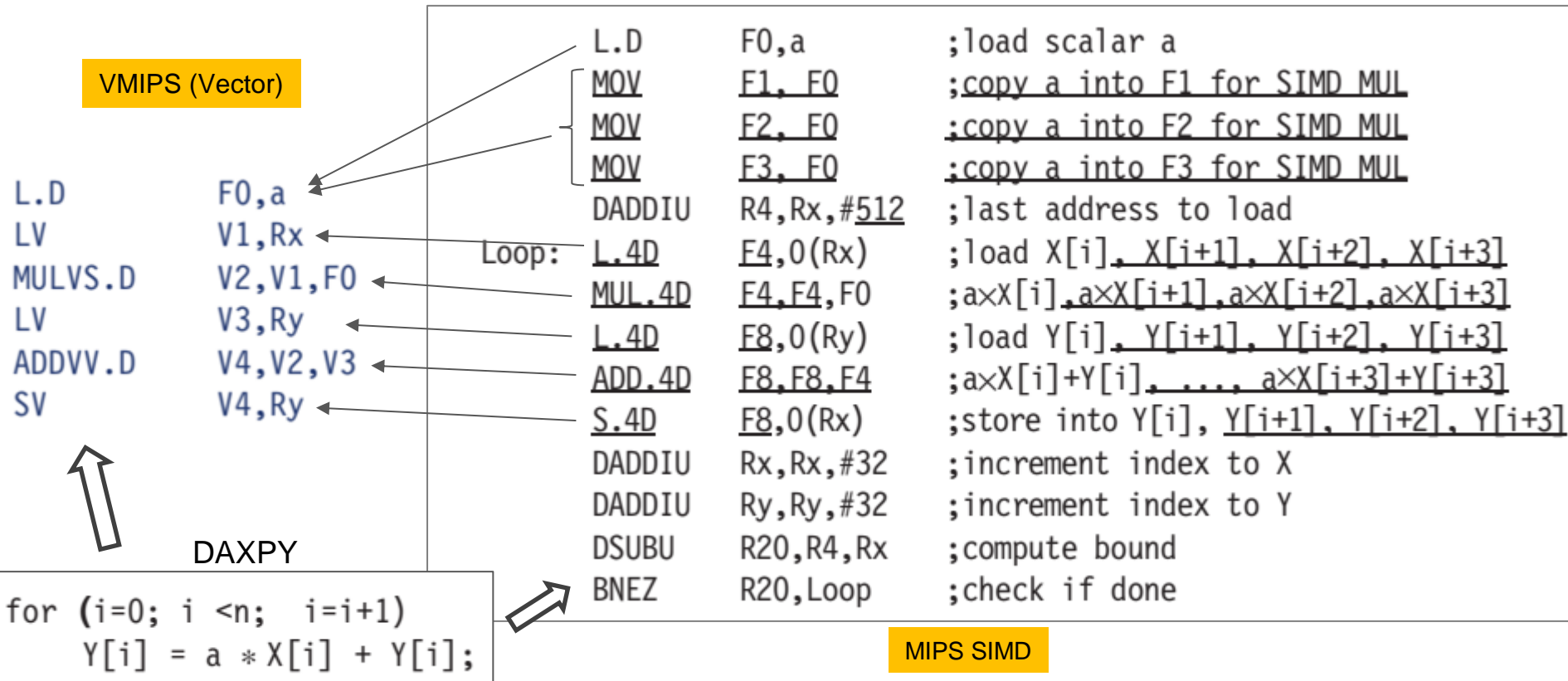




# Vector Machine vs. Multimedia SIMD

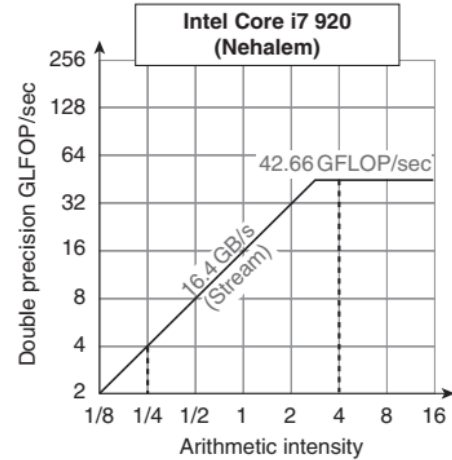
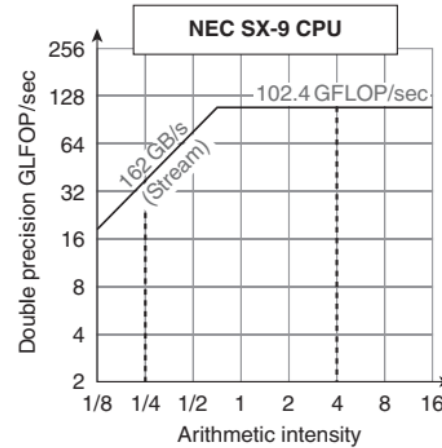
	<b>Vector Machine</b>	<b>Multimedia SIMD</b>
Register File	Large number of registers	Limited
Operand size	Variable size, indicated by VLR	Fixed (8, 16, 32, 64, 128, 256, 512)
Addressing modes	Striding, gather-scatter	NA
Conditional execution	Mask register support	NA

# Vector vs SIMD Multimedia Instructions



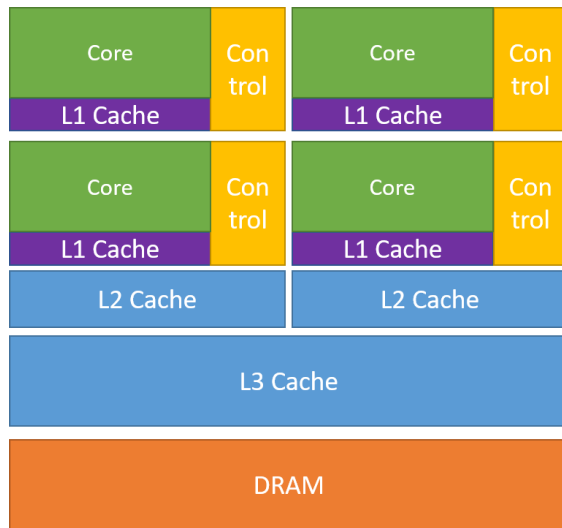
# Vector Machine vs Multimedia SIMD

- Arithmetic intensity =  $\frac{\text{No. of floating point ops}}{\text{Bytes transferred to mem}}$
- NEC SX-9 -> Vector machine
- Core i7 - Multimedia SIMD
- NEC SX-9
  - 10x faster at A.I of 0.25
  - 2x faster at A.I of 4



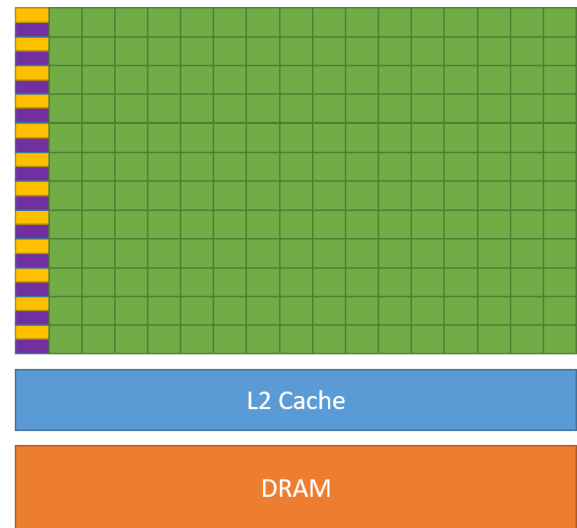
# GPU Architecture (DLP focus)

- Traditional GPU roles
  - Display output (HDMI, DVI, DisplayPort, etc)
  - Rendering (2D, 3D rendering, texture mapping, ray tracing, shaders, polygons etc)
  - Media encode/decode (MPEG, VP8, etc)
- CUDA and OpenCL
  - Opening up GPU's vector cores for general computing
- More complex than Vector processor or Multimedia SIMD



CPU

(Host)



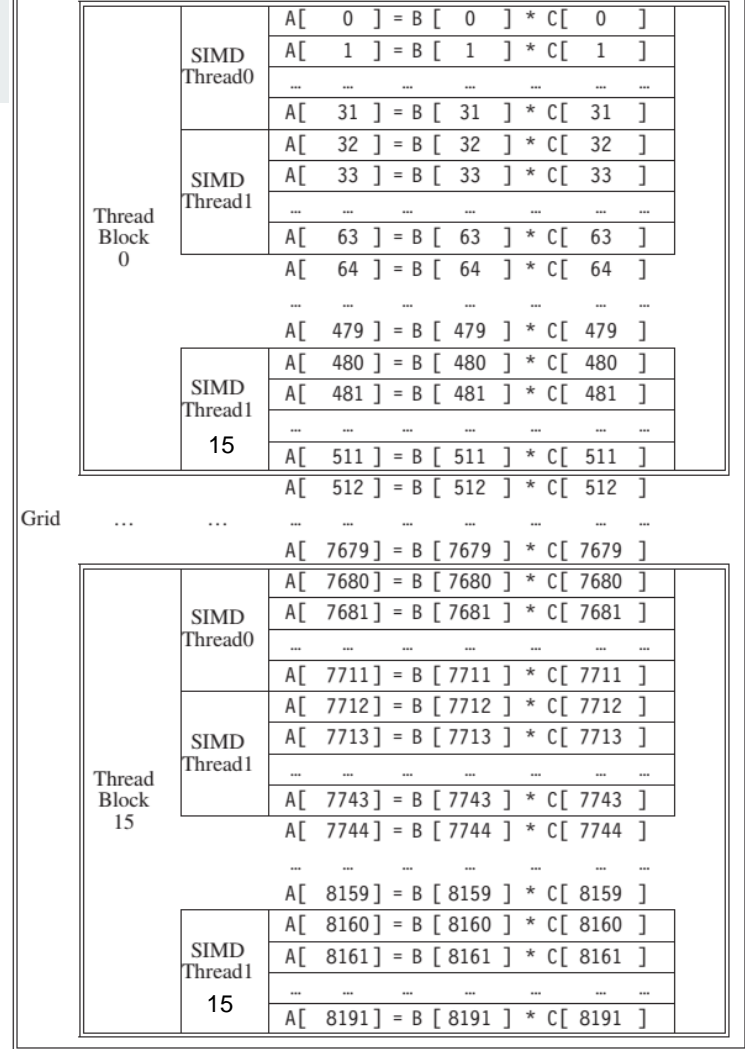
GPU

(Device)

# CUDA programming model

- Compute Unified Device Architecture (CUDA)
- Abstraction of GPU – thread, thread block, grid
- Thread – set of SIMD instructions that operate on a single lane (one unit of the loop)
- Thread block – vectorized loop consisting of multiple threads. Has shared memory
- Grid – the complete vectorized loop, consists of multiple thread blocks
- Kernel – part of code that runs on the GPU

```
for (int i=0; i<8192; i++)
    A[i] = B[i] * C[i];
```



# CUDA programming model

- Heterogeneous programming model. Uses CUDA toolkit (nvcc compiler)
- `__device__` or `__global__` -> GPU execution
- `__host__` -> CPU execution (C++ language extensions)
- `dimGrid` – dimension of Grid, `dimBlock` – dimension of block
- Kernel signature: `name<<<dimGrid, dimBlock>>>(... parameter list ...)`
- To each function: `blockIdx`, `threadIdx`, `blockDim` CUDA keywords available

```
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

CPU execution

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
    daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

GPU (CUDA) execution

# PTX – Parallel Thread Execution – Nvidia CUDA ISA

PTX instruction format:    `opcode.type d, a, b, c;`

```
shl.u32 R8, blockIdx, 9    ; Thread Block ID * Block size (512 or 29)
add.u32 R8, R8, threadIdx ; R8 = i = my CUDA Thread ID
shl.u32 R8, R8, 3          ; byte offset
ld.global.f64 RD0, [X+R8] ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8] ; RD2 = Y[i]
mul.f64 RD0, RD0, RD4      ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 RD0, RD0, RD2      ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0 ; Y[i] = sum (X[i]*a + Y[i])
```

CUDA Example – Matrix Multiplication

- [https://github.com/NVIDIA/cuda-samples/blob/master/Samples/0\\_Introduction/matrixMul/matrixMul.cu](https://github.com/NVIDIA/cuda-samples/blob/master/Samples/0_Introduction/matrixMul/matrixMul.cu)



# Vectorization

- Software Vectorization vs. Text Vectorization in NLP (vs. other vectorization..)
- Software Vectorization
  - Manual – using Intrinsic
  - Auto-vectorization – compile time and runtime
- Intel SIMD Intrinsic guide -  
<https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>
  - “Intrinsic” – C function that exposes underlying Assembly instruction
  - Examples: <https://github.com/Triple-Z/AVX-AVX2-Example-Code>



## Manual vectorization (FMA: $d = a * b + c$ )

```
multiply_and_add(const float* a, const float* b, const float*
c, float* d) {

    for(int i=0; i<8; i++) {
        d[i] = a[i] * b[i];
        d[i] = d[i] + c[i];
    }
}
```

Scalar Code

```
__m256 multiply_and_add(__m256 a, __m256 b,
__m256 c) {
    return _mm256_fmadd_ps(a, b, c);
}
```

Vectorized Code (AVX2)



# Loop Dependence (Hennessey 4.5)

- Loop Dependences
  - Data Dependences (Loop Carried Dependence)
    - Forward Dependence
    - Backward Dependence
    - Cyclic Dependence
  - Name Dependences
    - Resolved by renaming
- Compiler vectorization
  - Analyses code and identifies Loops, Array references, Indirections, Induction variables

# Loop dependences identification

## Scalar code (Ex: 1)

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

$x[i] = x[i] + s;$

$= x[2] + s, \dots$

## Parallel execution (imagine..)

$y[0] = x[0] + s, \quad y[1] = x[1] + s, \quad y[2]$

- No loop carried dependence (i.e no dependence across iterations)
- But it has dependence due to induction variable

# Loop dependences identification

Ex: 2

```
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 */
```

```
}
```

Analysis

- S1 -> has **loop carried dependence** because  $A[i + 1]$  needs  $A[i]$  calculated from previous iteration (cyclic)
- S2 -> has **loop carried dependence** because  $B[i + 1]$  needs  $B[i]$  calculated from previous iteration (cyclic)
- S2 ->  $B[i+1]$  **depends** on S1 which computes  $A[i+1]$ . But this is not loop-carried. But it is a True Dependence

*Parallel Execution*

$A[1] = A[0] + C[0]$

$A[2] = A[1] + C[1]$

$A[3] = A[2]$

$B[1] = B[0] + A[1]$

$B[2] = B[1] + A[2]$

$B[3] = B[2] + A[3]$

$B[4] = B[3] + A[4]$

...

# Loop dependences removal

Ex: 3

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

Analysis

- No circular dependency
- Loop carried dependence in S1  
-> requires B[i] calculated from S2 in previous loop
- S2 does not depend on S1
- S2 has dependency on B[0]

# Loop dependences removal

Ex: 3

```
for (i=0; i<100; i=i+1) {
```

```
    A[i] = A[i] + B[i];    /* S1 */
```

```
    B[i+1] = C[i] + D[i]; /* S2 */
```

```
}
```



After Loop Dependency removal

```
A[0] = A[0] + B[0];
```

```
for (i=0; i<99; i=i+1) {
```

```
    B[i+1] = C[i] + D[i];
```

```
    A[i+1] = A[i+1] + B[i+1];
```

```
}
```

```
B[100] = C[99] + D[99];
```

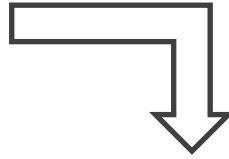
# Name Dependences identification in Loop

```
for (i=0; i<100; i=i+1) {  
    Y[i] = X[i] / c; /* S1 */  
    X[i] = X[i] + c; /* S2 */  
    Z[i] = Y[i] + c; /* S3 */  
    Y[i] = c - Y[i]; /* S4 */  
}
```

- True dependences (RAW) from S1 to S3 and from S1 to S4 because of Y[i].
  - These dependences will force S3 and S4 to wait for S1 to complete.
- Anti-dependence (WAR) from S1 to S2, based on X[i].
- Anti-dependence (WAR) from S3 to S4 for Y[i].
- Output dependence (WAW) from S1 to S4, based on Y[i].

# Name Dependences removal in Loop

```
for (i=0; i<100; i=i+1) {  
    Y[i] = X[i] / c; /* S1 */  
    X[i] = X[i] + c; /* S2 */  
    Z[i] = Y[i] + c; /* S3 */  
    Y[i] = c - Y[i]; /* S4 */  
}
```



```
for (i=0; i<100; i=i+1 {  
    T[i] = X[i] / c; /* Y renamed to T to remove output dependence */  
    X1[i] = X[i] + c; /* X renamed to X1 to remove antidependence */  
    Z[i] = T[i] + c; /* Y renamed to T to remove antidependence */  
    Y[i] = c - T[i];  
}
```



## Other Dependences

- Recurrence – type of loop carried dependence where value depends on previous iteration

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

- Vector processors have special instructions to handle this.
- Compiler builds dependency graphs to identify and resolve dependences
- Compiler Optimization – vast and active research area (Auto-vectorization)

- Affine indices:
  - Algorithms work on affine indices
  - $a \times i + b$  (a and b are loop constants)
- Non-affine index
  - Indirect reference, as in sparse arrays
- GCD test typically used to detect dependency
  - For two indices of the form  $a \times j + b, c \times k + d$ ,
  - If  $\text{GCD}(c, a)$  divides  $(d-b)$ , LD exists
- Loop dependence detection is an NP-complete problem!



# Vectorization Performance (Examples)

- NumPy – Numeric processing package for Python
- Pandas – Data analysis tool for Python
- Both implement SIMD Vectorization for arrays
- Extensively used in Machine learning
  
- Classical machine learning – SIMD vectorization usually sufficient
- Deep learning – GPU used extensively (CUDA)
  - PyTorch
  - TensorFlow
  
- Examples from:
  - <https://medium.com/codex/say-goodbye-to-loops-in-python-and-welcome-vectorization-e4df66615a52>

# NumPy: Speedup of Vector Sum calculation

```
import time
start = time.time()

# iterative sum
total = 0
# iterating through 1.5 Million numbers
for item in range(0, 1500000):
    total = total + item

print('sum is:' + str(total))
end = time.time()

print(end - start)
```

```
#1124999250000
#0.14 Seconds
```

## Vectorized (~18x faster)

```
import numpy as np

start = time.time()

# vectorized sum - using numpy for vectorization
# np.arange create the sequence of numbers from 0 to 1499999
print(np.sum(np.arange(1500000)))

end = time.time()

print(end - start)
```

```
##1124999250000
##0.008 Seconds
```

# NumPy: Speedup of Vector Dot Product calculation

## Vectorized (~165x faster)

```
total = 0
tic = time.process_time()

for i in range(0,5000000):
    total = 0
    for j in range(0,5):
        total = total + x[i][j]*m[0][j]

    zer[i] = total

toc = time.process_time()
print ("Computation time = " + str((toc - tic)) + "seconds")

####Computation time = 28.228 seconds
```

```
tic = time.process_time()

#dot product
np.dot(x,m.T)

toc = time.process_time()
print ("Computation time = " + str((toc - tic)) + "seconds")

####Computation time = 0.107 seconds
```

# Pandas: Speedup of Vector ratio calculation

```
import time
start = time.time()

# Iterating through DataFrame using iterrows
for idx, row in df.iterrows():
    # creating a new column
    df.at[idx, 'ratio'] = 100 * (row["d"] / row["c"])
end = time.time()
print(end - start)
### 109 Seconds
```

**Vectorized (~1000x faster)**

```
start = time.time()
df["ratio"] = 100 * (df["d"] / df["c"])

end = time.time()
print(end - start)
### 0.12 seconds
```



# Arithmetic Intensity Estimation

- Covered in Intel® Advisor topic..



# Recap

- Concepts
  - Vectorization, Auto-vectorization
  - SIMD example – Numpy, SIMD intrinsics
  - Vector vs SIMD vs GPU
  - GPU example
- Problems
  - Arithmetic Intensity estimation
  - Loop dependency identification and removal
  - Indexed Vector Add
  - Stride calculation