TEXAS STATE UNIVERSITY
The rising STAR of Texas

Intra-Processor Parallel Architecture

Course No
Lecture No

Term

Module developed Spring 2015
by Dan Tamir

This module created with support form NSF under grant # DUE 1141022

Some slides adopted from Patterson and Hennessy 4th Edition with permission

Outline

Intra Core Parallelism

- Amdahl's law
- Parallelism Taxonomy
- Data Level Parallelism
- Instruction Level Parallelism
  - "Simple" (Naïve) Example
  - Realistic Example

AMDAHL'S LAW

Amdahl's Law and Parallelism

Speedup with parallelism = 1 / ((1-F) + F/P)

- Assume we can parallelize 25% of the program and we have 20 processing cores

Speedup w/ E = 1/(.75 + .25/20) = 1.31

- If only 15% is parallelized

Speedup w/ E = 1/(.85 + .15/20) = 1.17

- Amdahl's Law tells us that to achieve linear speedup with n processors, none of the original computation can be scalar!
- To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less

Speedup w/ E = 1/(.001 + .999/100) = 90.99

Amdahl's Law and Parallelism

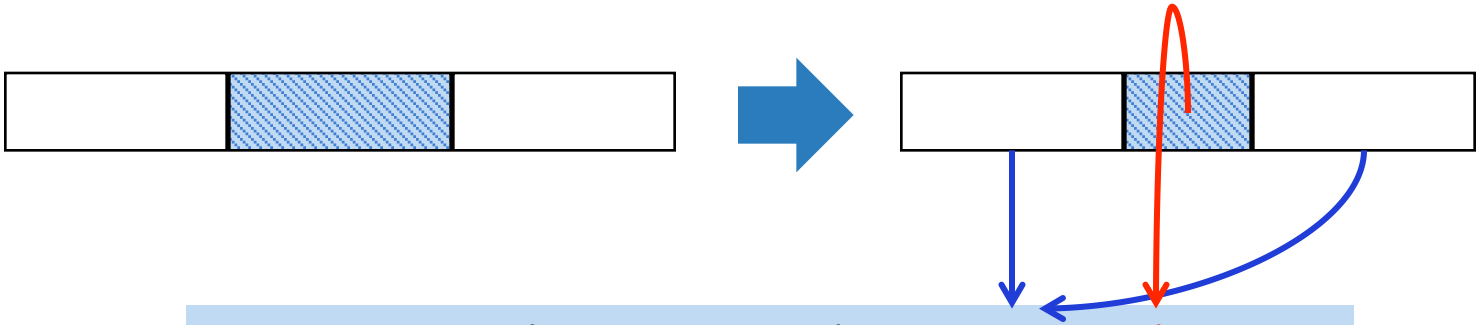
Speedup due to enhancement E is

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

Gene Amdahl

Suppose,

Fraction F can be parallelized into P processing cores



ExTime w/ E = ExTime w/o E × ((1-F) + F/P)

Speedup w/ E = 1 / ((1-F) + F/P)

PARALLELISM TAXONOMY

Flynn's Taxonomy

- SISD – single instruction, single data stream
  - aka uniprocessor
- SIMD – single instruction, multiple data streams
  - single control unit broadcasting operations to multiple datapaths
- MISD – multiple instruction, single data
  - no such machine
- MIMD – multiple instructions, multiple data streams
  - aka multiprocessors (SMPs, clusters)

Types of Parallelism

- Models of parallelism
  - Message Passing
  - Shared-memory
- Classification based on decomposition
  - Data parallelism
  - Task parallelism
  - Pipelined parallelism
- Classification based on
  - TLP // thread-level parallelism
  - ILP // instruction-level parallelism
- Other related terms
  - Massively Parallel
  - Petascale, Exascale computing
  - Embarrassingly Parallel

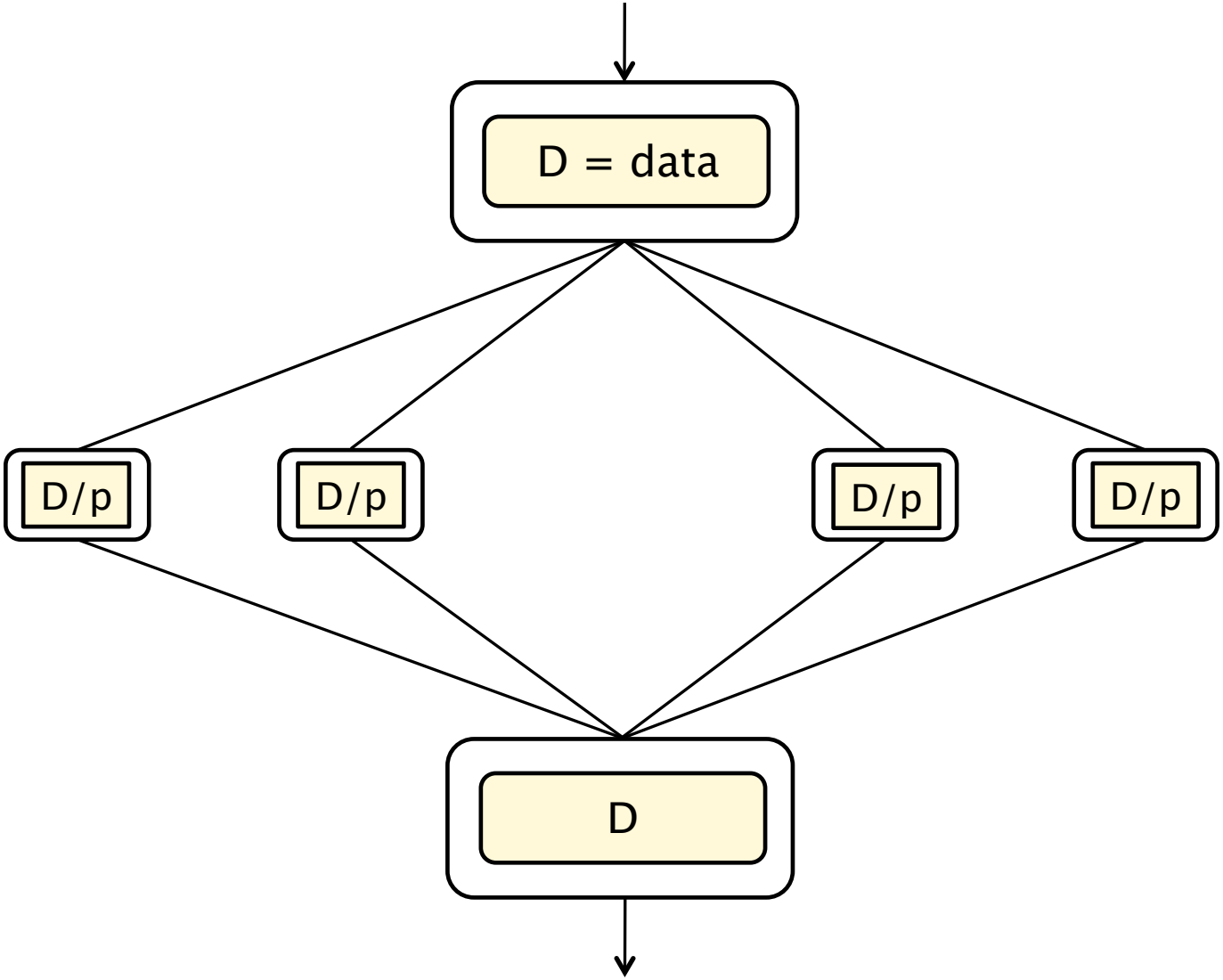
Types of Parallelism

- Models of parallelism
  - Message Passing
  - Shared-memory
- Classification based on decomposition
  - Data parallelism
  - Task parallelism
  - Pipelined parallelism
- Classification based on
  - TLP // thread-level parallelism
  - ILP // instruction-level parallelism
- Other related terms
  - Massively Parallel
  - Petascale, Exascale computing
  - Embarrassingly Parallel

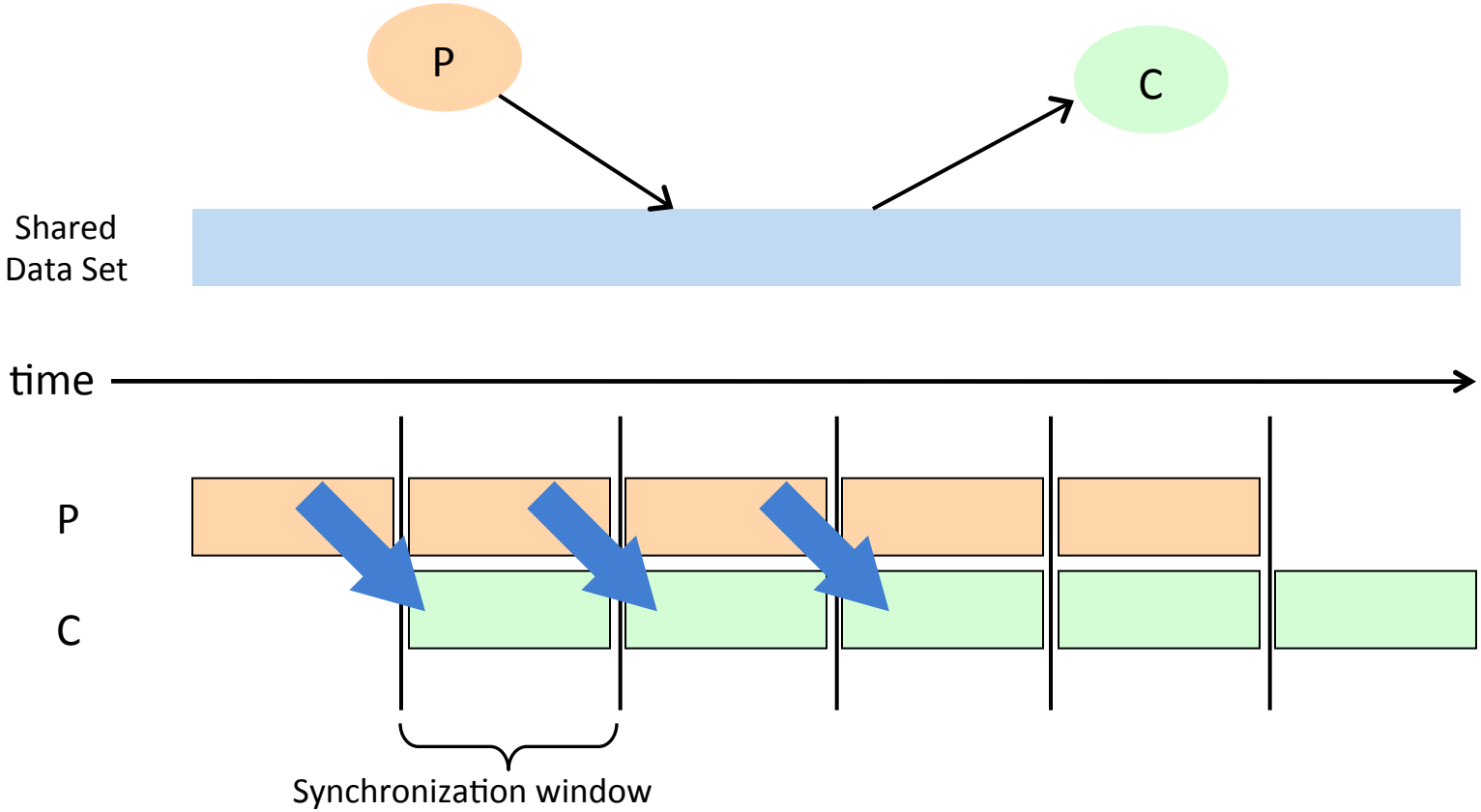
Data Parallelism

- This type of parallelism is sometimes referred to as loop-level parallelism
- Quite common in scientific computation
- Also, sorting. Which ones?

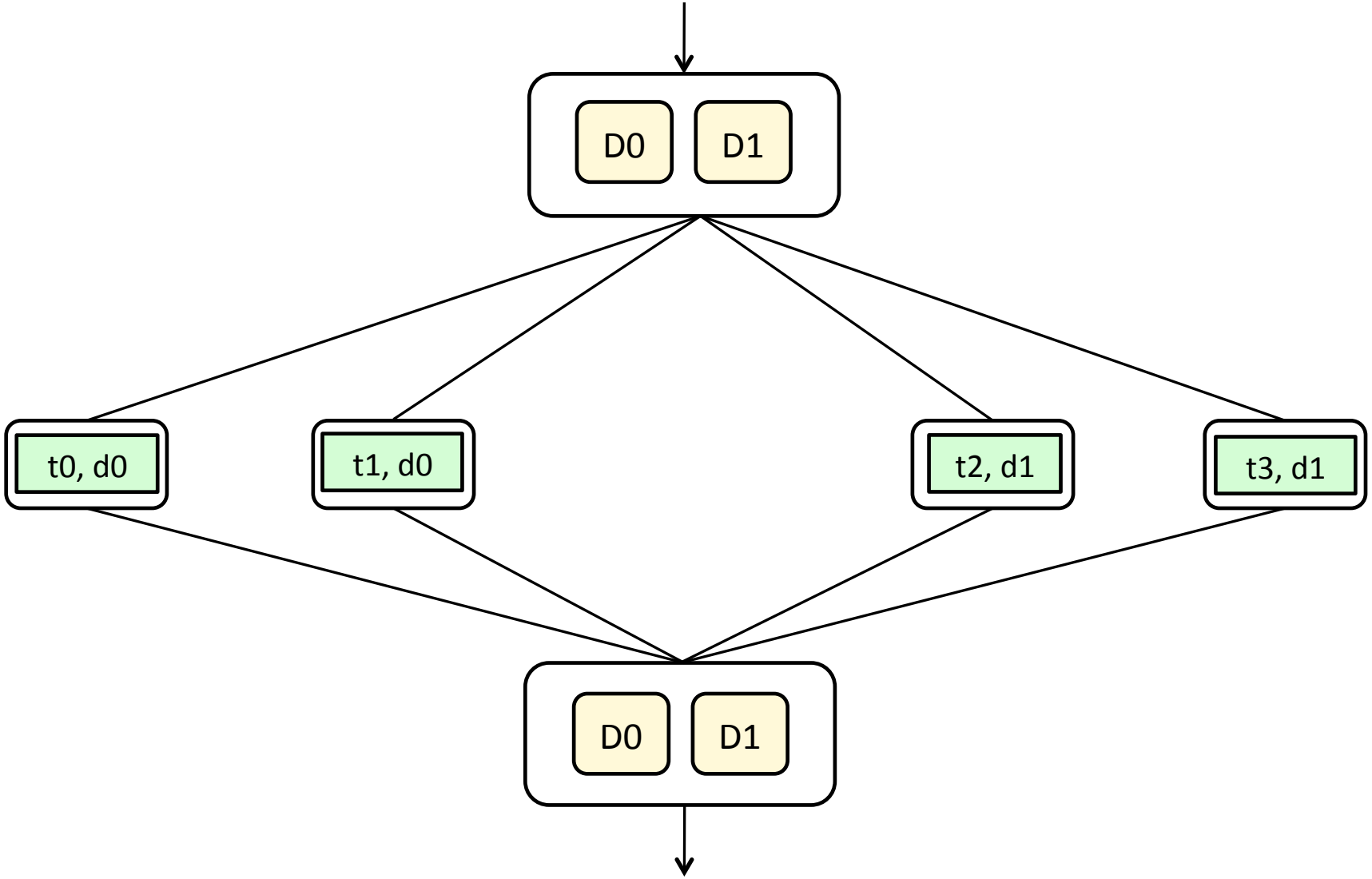
Data Parallelism



Pipelined Parallelism



Task Parallelism



DATA LEVEL PARALLELISM

Data Level Parallelism

- DLP might be exploited when the same set of operations is repeatedly applied to data
  - For example, in image processing we might apply the same set of operations to every pixel of the image $I1$ and the image $I2$ to obtain an image $I3$.
- Assuming that operations on individual image pixels are independent, then several mechanisms can be used to parallelize these operations
- One relatively simple option is to use a SIMD approach, where the SIMD is applied at the instruction level
- Intel MMX, Motorola-Apple Altivec are examples of DLP at the instruction level

DLP/SIMD at the Instruction Level

- A simple DLP, via SIMD at the instruction level, uses the fact that current ALUs operate on long words (registers) yet some data structures require short words
- Often register length is 128/64/32 bit but the data structure requires entities of 32/16/8 bits
- For example image processing is often performed at the 8-bit pixel level.

DLP/SIMD-ISA

- Idea – break the long word into several short words
- Redesign the execution units to perform the same operation in parallel on the short entities.
- We first look at an example then analyze it.

Example (C++ Code)

```
int i = 99;
int A[100], B[100], C[100];
while (i>=0)
{
        A[i]=B[i]+C[i];
        i=i-1;
}
```

MIPS Snippet

```
.data
A:      .space  400                             #allocate 400 bytes (100 integers)
B:      .space  400                             #allocate 400 bytes (100 integers)
C:      .space  400                             #allocate 400 bytes (100 integers)
.text
        la      $t0,    A                       #$t0 points to A[0]
        la      $t2,    B                       #$t2 points to B[0]
        la      $t4,    C                       #$t4 points to C[0]
        li      $s0     99                      #$s0 is the counter (i)
wlp1s:  lw      $t1,    396($t0)                #$t1=A[i]
        lw      $t3     396($t2)                #$t3=B[i]
        add     $t5,    $t1,    $t3             #$t5=A[i]+B[i]
        sw      $t5     396($t4)                #C[i]=$t5
        subi    $t0,    $t0,    4               # decrement pointer to A
        subi    $t2,    $t2,    4               #decrement pointer to B
        subi    $t4,    $t4,    4               #decrement pointer to C
        subi    $s0,    $s0,    1               #decrement counter (i)
        bgez    $s0,    wlp1s
```

MIPS Snippet (initialization)

```
.data
A:  .space      400     #allocate 400 bytes (or space for 100 integers)
B:  .space      400     #allocate 400 bytes (or space for 100 integers)
C:  .space      400     #allocate 400 bytes (or space for 100 integers)

.text
    la    $t0,    A         #$t0 points to A ($t0=&A[0])
    la    $t2,    B         #$t2 points to B
    la    $t4,    C         #$t4 points to C
    li    $s0     99        #$s0 is the counter (i)
```

MIPS Snippet (loop)

```
wlp1s:  lw      $t1,    396($t0)        #$t1=A[i]
        lw      $t3     396($t2)        #$t3=B[i]
        add     $t5,    $t1,    $t3     #$t5=A[i]+B[i]
        sw      $t5     396($t4)        #C[i]=$t5
        subi    $t0,    $t0,    4       #decrement pointer to A
        subi    $t2,    $t2,    4        # decrement pointer to B
        subi    $t4,    $t4,    4        # decrement pointer to C
        subi    $s0,    $s0,    1       #decrement counter (i)
        bgez    $s0,    wlp1s           #branch back
```

- Total of 100 iterations

C++ Code with unsigned bytes ("pixels")

```
int i = 99;
unsigned char A[100], B[100], C[100];
while (i>=0)
{
        A[i]=b[i]+C[i];
        i=i-1;
}
```

MIPS Snippet (initialization)

```
        .data
A:      .space      100     #allocate 100 bytes (or space for 100 “pixels”)
B:      .space      100     #allocate 100 bytes (or space for 100 “pixels”)
C:      .space      100     #allocate 100 bytes (or space for 100 “pixels”)

        .text
        la      $t0,    A       #$t0 points to A
        la      $t2,    B       #$t2 points to B
        la      $t4,    C       #$t4 points to C
        li      $s0     99      #$s0 is the counter (i)
```

MIPS Snippet (loop)

```
wlp1s: lbu   $t1,  99($t0)          #$t1=A[i]
       lbu   $t3   99($t2)          #$t3=B[i]
       add   $t5,  $t1,   $t3   #$t5=A[i]+B[i]
       sb    $t5   99($t4)          #C[i]=$t5
       subi  $t0,  $t0,   1     #decrement pointer to A
       subi  $t2,  $t2,   1      # decrement pointer to B
       subi  $t4,  $t4,   1      # decrement pointer to C
       subi  $s0,  $s0,   1     #decrement counter (i)
       bgez  $s0,  wlp1s        #branch back
```

- Total of 100 iterations

Imaginary SIMD Extension of MIPS

- Alice is an engineer in the MIPS company
- She is tasked with accelerating the performance of MIPS with respect to image processing
- Alice talks with the design team and finds out that it would be relatively inexpensive to introduce the instruction “add4” to the MIPS micro-architecture
- Alice writes pseudo SIMD-MIPS code to convince stake-holders that adding this instruction will speed-up some image processing operations.
- She plans to use this example to explore adding a SIMD extension to the MIPS

C++ Code with "pixels"

```
int i = 100;
unsigned char A[100], B[100], C[100];
while (i>0)
{
        A[i]=b[i]+C[i];
        i=i-1;
}
```

SIMD-MIPS Snippet (initialization)

```
        .data
A:      .space      100     #allocate 100 bytes (or space for 100 "pixels")
B:      .space      100     #allocate 100 bytes (or space for 100 "pixels")
C:      .space      100     #allocate 100 bytes (or space for 100 "pixels")

        .text
        la      $t0,    A       #$t0 points to A
        la      $t2,    B       #$t2 points to B
        la      $t4,    C       #$t4 points to C
        li      $s0     24      #$s0 is the counter (i)
```

SIMD-MIPS Snippet (loop)

```
wlp1s:  lw      $t1,    96($t0)         #$t1=A[i;i+1;i+2;i+3]
        lw      $t3     96($t2)         #$t3=B[i;i+1;i+2;i+3]
        add4    $t5,    $t1,    $t3

                                        #$t5=A[i;i+1;i+2;i+3]+B[i;i+1;i+2;i+3]

        sw      $t5     96($t4)         #C[i]=$t5
        subi    $t0,    $t0,    4       #decrement pointer to A
        subi    $t2,    $t2,    4        # decrement pointer to B
        subi    $t4,    $t4,    4        # decrement pointer to C
        subi    $s0,    $s0,    1       #decrement counter (i)
        bgez    $s0,    wlp1s           #branch back
```

- 25 Iterations

Add4 Digest

- Implementing the add4 instructions might be relatively simple
- Relevant intermediate carry bits are disabled
- The same can be applied to add8 (for 64 bit registers) and add2 (e.g., two shorts)
- The name add4 is not a good mnemonic it is used here for simplification
  - Better use addbyte,addshort etc.

Add4 – low level illustration

- Consider the snippet:

```
li      $t1     0xFCFDFEFF
li      $t2     0x01010101
add     $t0,    $t1,    $t2             #$t0=0xFDFF0000
```

- Now consider the snippet:

```
li      $t1     0xFCFDFEFF
li      $t2     0x01010101
add4    $t0,    $t1,    $t2             #$t0=0xFDFEFF00
```

- Each byte is incremented by 1. There is an overflow in the LS-Byte. The carry does not propagates beyond the byte boundary.

Performance Analysis

- There is a 4x speedup in the loop.
- Generally, there would be some overhead inside loops – loop control
  - hence a speedup of 3.6-3.8 is an excellent achievement.
- Generally, there would be some overhead outside loops and the program might include parts that cannot be parallelized
  - Hence speedup is limited according to Amdahl's law
  - Identifying and utilizing instruction level DLP by the compiler and/ or the programmer is not trivial

Multimedia SIMD Extensions

- Most processors today come with vector extensions
- The compiler needs to be able generate code for these hardware units

- Examples
  - Intel : MMX, SSE
  - AMD : 3DNow!
  - IBM, Apple : Altivec (does floating-point as well)

Intel MMX

Fetched from:

http://homepages.cae.wisc.edu/~ece734/mmx/24308102.pdf

The four MMX technology data types are:

- Packed byte — Eight bytes packed into one 64-bit quantity
- Packed word — Four 16-bit words packed into one 64-bit quantity
- Packed doubleword — Two 32-bit double words packed into one 64-bit quantity
- Quadword — One 64-bit quantity

Intel MMX

The MMX instructions cover several functional areas including:

- Basic arithmetic operations such as add, subtract, multiply, arithmetic shift and multiply-add
- Comparison operations
- Conversion instructions to convert between the new data types - pack data together, and unpack from small to larger data types
- Logical operations such as AND, AND NOT,OR, and XOR
- Shift operations
- Data Transfer (MOV) instructions for MMX register-to-register transfers, or 64-bit and 32-bit load/store to memory

Vector Processors

- A vector processor operates on **vector registers**
- Vector registers can hold multiple data items of the same type
  - Usually 64, 128 words
- Need hardware to support operations on these vector registers
- Need at least one regular CPU
- Vector ALUs usually pipelined
  - high pay-off on scientific applications
  - formed the basis of supercomputers in the 1970's and early 80's

Example Vector Machines

	Maker	Year	Peak perf.	# vector Processors	PE clock (MHz)
STAR-100	CDC	1970	??	113	2
ASC	TI	1970	20 MFLOPS	1, 2, or 4	16
Cray 1	Cray	1976	80 to 240 MFLOPS		80
Cray Y-MP	Cray	1988	333 MFLOPS	2, 4, or 8	167
Earth Simulator	NEC	2002	35.86 TFLOPS	8	

Summary and Conclusions (DLP)

- A method for exploiting parallelism at the data level has been presented and analyzed
- An example using MIPS ISA and an imaginary SIMD extension to MIPS is used
- Speedup is briefly analyzed
- It might be difficult for the compiler to exploit this level of parallelism

INSTRUCTION LEVEL PARALLELISM

Instruction Level Parallelism

- ILP might be exploited when a set of instructions can be executed in parallel
  - This can be enabled by introducing multitudes of resources (e.g., four ALUs)
  - Multithreading can increase the exploitable ILP

Instruction Level Parallelism

- Let us start with an example.
- Consider the following code snippet:

```
.
.
.
i:      add     $t0,    $t1,    $t2
j:      add     $t3,    $t4,    $t5
k:      add     $t6,    $t3,    $t7
.
.
.
```

- Theoretically, the instructions at labels, i and j, can be executed in parallel.
- On the other hand, the instructions j and k do not seem to be suitable for parallel execution. Why?
- What are the requirements for executing the instructions (i), (j) in parallel?
- Note that this is similar to Data Hazards in Pipeline and the remedy might be similar

Duplicating Compute Resource

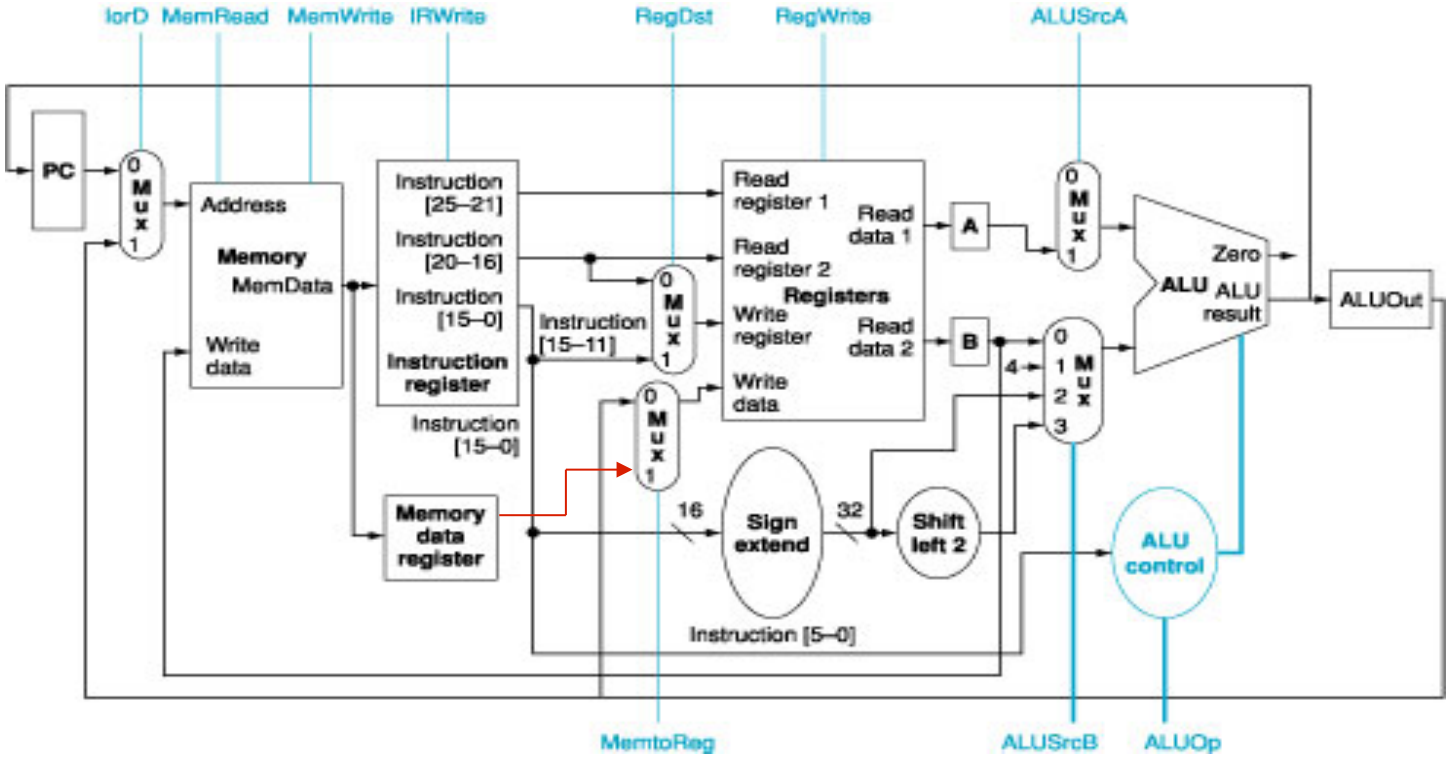
- Executing an instruction such as:

  i: add $t0, $t1, $t2 (and other MIPS

  Instructions) requires compute resources such as an ALU, load /store unit, pipeline, etc.

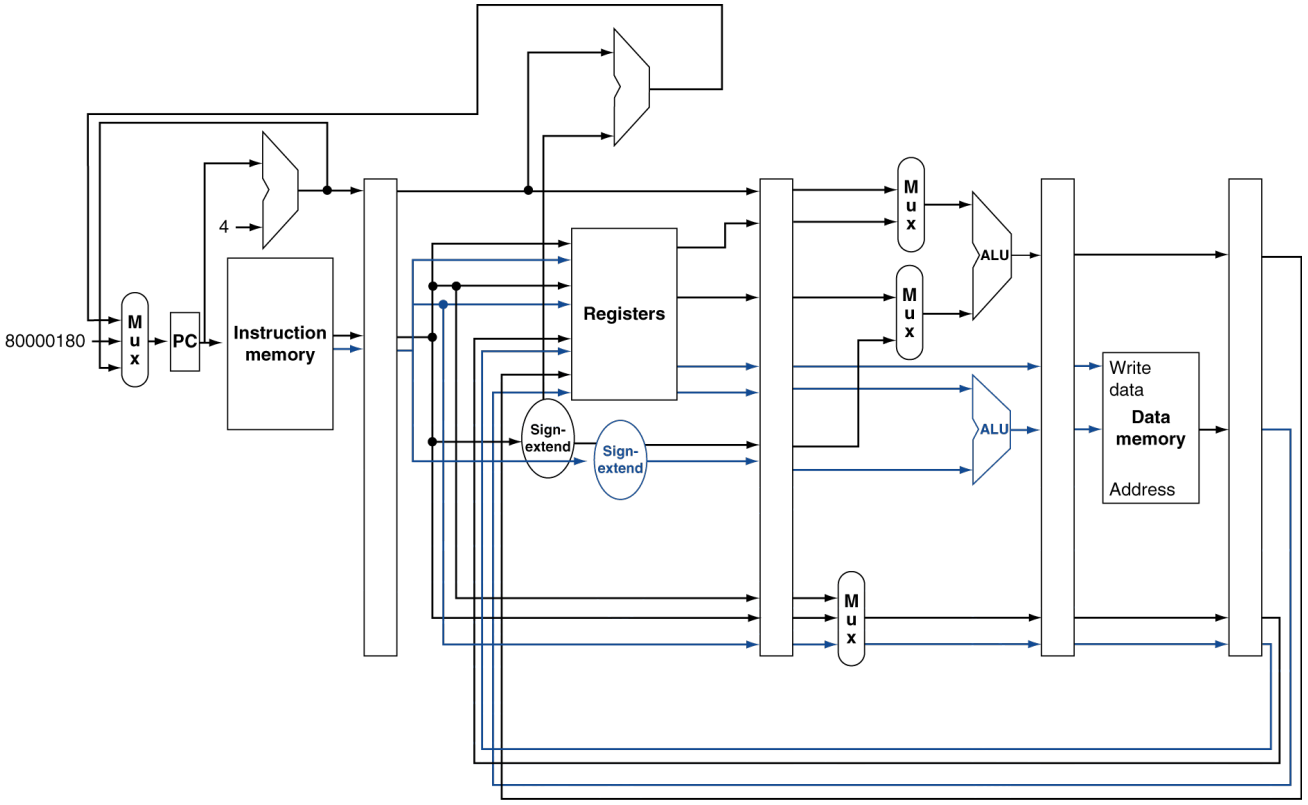
- Duplicating resources. For example using two ALUs might enable executing two ALU instructions in Parallel.
- This is feasible given the large degree of integration with millions of transistors available on the die.
  - might be cost effective

A simplified MIPS Architecture Diagram



MIPS Architecture Diagram

An Imaginary Dual ALU MIPS Architecture Diagram



(Imaginary) Dual Issue MIPS Diagram

Instruction Dependency

- There are several forms of instruction dependency.
- Some of these forms pose a significant challenge for exploiting ILP even in the presence of additional resources
- We will concentrate on one form of dependency
  - source-destination dependency
- Instructions j and k exhibit source- destination dependency if one of the sources of instruction j is the destination of instruction k.
- This is indeed the case in the example given before

Source-Destination Dependency

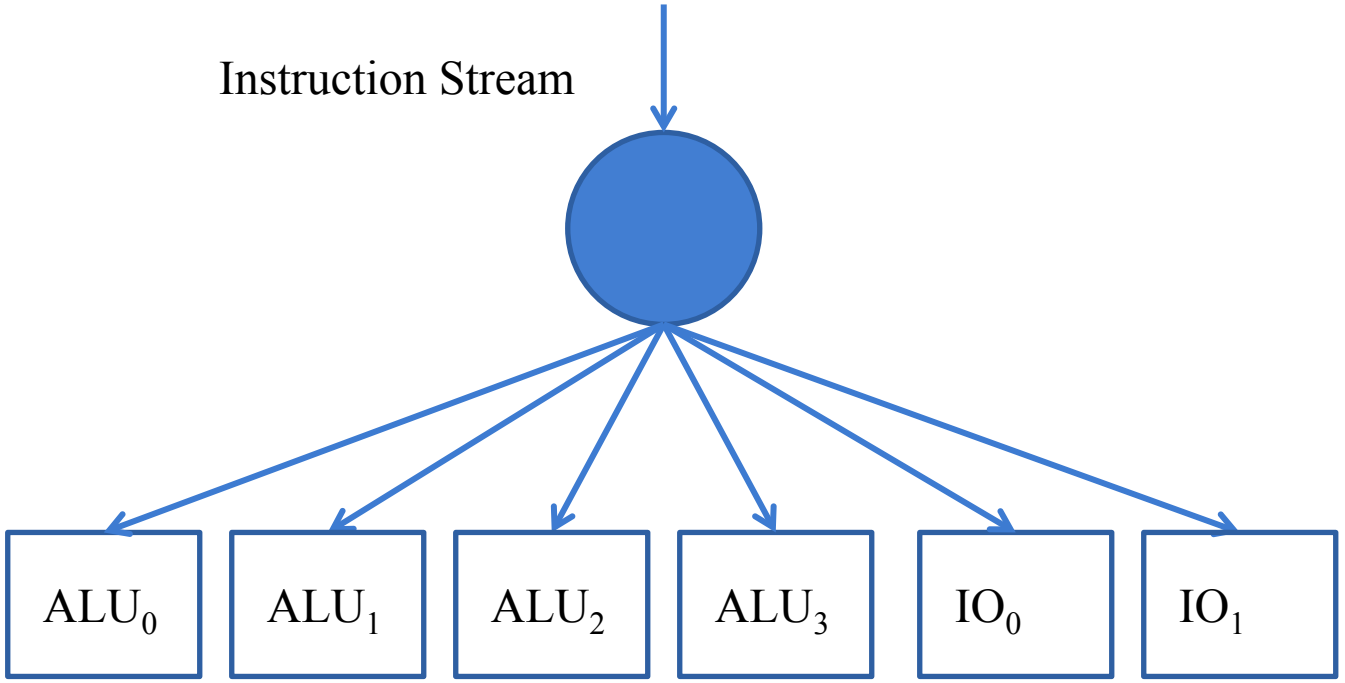
- Consider:

```
i:      add     $t0,    $t1,    $t2
j:      add     $t3,    $t4,    $t5
k:      add     $t6,    $t3,    $t7
```

- In this example, $t3 is the destination of instruction j, and a source for instruction k.
- Hence instructions j and k exhibit a source-destination dependency.
- Intuitively the execution of j has to be completed before the execution of k starts
- Practically, there are still ways to partially overlap the executions of j and k
- There are several dependencies posing different type of parallelism challenges
- In the rest of the discussion we will only consider source-destination dependency and assume that instructions with source destination dependency cannot be executed in parallel

An Imaginary ILP enhanced MIPS

- Consider a 6-issue MIPS.
- The architecture has four ALUs and two I/O unit



ILP-MIPS Programming

- The ILP enhanced MIPS can execute in parallel instructions such as:

  add $t3, $t0, $t1|| add $t4, $t0, $t1 || sw $t3, 0($t2)||
  sw $t4, 4($t2)|| addi $t2, $t2, 4 || addi $so, $s0,1

- Next we will examine the potential of this machine via example

Very simple example of ILP

Example (C++ Code)

```
int i = 0, y[100], X0=alpha, X1=beta;
While(i<100)
{
        y[i]=X0+X1;
        i=i+1;
}
```

MIPS Snippet (Initialization)

```
.data
y:      .space  400             #allocate 400 bytes (100 integers)
X0:     .space  4       #assume that alpha is stored here
X1:     .space  4       #assume that beta is stored here
.
.text
.
        la      $t0,    X0              #$t0 points to X0
        lw      $t0,    0($t0)          #$t0 contains  X0 i.e., alpha
        la      $t1,    X1              #$t1 points to X1
        lw      $t1,    0($t1)          #$t1 contains  X1 i.e., beta
        la      $t2,    y               #$t2 points to y[0]
        li      $s0     0               #$s0 is the counter (i)
        li      $s1     100             #Bound
```

MIPS Snippet (loop)

```
wlp1s: add   $t3,   $t0,    $t1     #$t3=X0+X1
       sw    $t3    0($t2)          #y[i]=$t3
       addi  $t2,   $t2,    4       #Increment pointer
       addi  $s0,   $s0,    1       # brunch back
       bne   $s0,   $s1,    wlpls   #brunch back
```

- Total of 100 iterations
- 500 instructions

A simple instance of loop unrolling

```
int i = 0, y[100], X0=alpha, X1=beta;
While(i<100)
{
        y[i]=X0+X1;
        i=i+1;
        y[i]=X0+X1;
        i=i+1;
}
```

Adding Pipelining to Better Fit ILP Based Assembly

```
int i = 0, y[100], X0=alpha, X1=beta;

        Y[i]= X0+X1; i=i+1; Y[i]= X0+X1; i=i+1;
        While(i<48)
                {y[i]=X0+X1;
                i=i+1;
                y[i]=X0+X1;
                i=i+1;}
        Y[i]= X0+X1; i=i+1; Y[i]= X0+X1;
```

MIPS Snippet (loop)

```
        add $t3, $t0, $t1|| add $t4, $t0, $t1 || sw $t3, 0($t2)||
        sw $t4, 4($t2)|| addi $t2, $t2, 4 || addi $so, $s0,1
wlp1s:
        add $t3, $t0, $t1|| add $t4, $t0, $t1 || sw $t3, 0($t2)||
        sw $t4, 4($t2)|| addi $t2, $t2, 4 || addi $so, $s0,1
        bne     $s0,    $s1,    wlpls   #brunch back

         add $t3, $t0, $t1|| add $t4, $t0, $t1 ||
        |w $t3, 0($t2)|| sw $t4, 4($t2)||
```

- Total of 48 iterations + 2 additional ILP instructions
- 100 Cycles

Performance Analysis (of the example)

- About 6x duplication of certain units (which occupy relatively small portion of the die)
- There is a 5x speedup in the loop.
- Generally, there would be some overhead inside loops including overhead for prologue, epilogue, and loop handling
  - hence a speedup of 3.6-3.8 for 4 ALUs is an excellent achievement.
- Generally, there would be some overhead outside loops and the program might include parts that cannot be parallelized
  - Hence speedup is limited according to Amdahl's law (will be studied in computer architecture)
- Identifying and utilizing instruction level ILP by the compiler and/ or the programmer is not trivial
  - Will be discussed later

Realistic Example of ILP

Example: Digital Filter

- A linear digital filter can be implemented via convolution expressed mathematically in the following form:

$$y_n = \sum_{i=0}^{p-1} a_i \times x_{n-i} \; ; \qquad n = p \ldots N$$

- In this case, x_i $(0 \le i \le n)$ is the i^{th} sample of the input signal x, y_i $(p \le i \le n)$ is the i^{th} sample of the output signal y, and a_i $(0 \le i \le p)$ is the i^{th} coefficient of the filter.

Digital Filter (C++ Code)

```
int a[10], x[100], y[100];
int p=10;

/* Assuming that a[] contains coefficients and x contains the signal */

for(i=0; i<100; i++)
y[i]=0;                          // Effectively zeroing the first p samples of y

for (n=p; n<100; n++)
    for (i=0; i<p; i++)
        y[n] += a[i]*x[n-i];
```

- In the following MIPS code example we assume that P=2 and perform loop unrolling for the Internal loop. This can save time and increase the ILP.

- The code is optimized for performance

MIPS Code

	.align 2				
	.data				
x:	.space	800			
					#space of x & y
	.text				
	la	$a3	x		#Assumes x, y are continuous
	sw	$0	400($s0)		# y[0]=0
	li	$a0	3		# coefficients
	li	$a1	2		
	addi	$a2	$a3	396	
	lw	$t1	0($a3)		# x[n-1]
For0:	bge	$a3	$a2	EndFor0	
	lw	$t2	4($a3)		# x[n]
	mul	$t1	$t1	$a0	$t1=x[n]*a0
	mul	$t2	$t2	$a1	$t2=x[n-1]*a1
	add	$t3	$t1	$t2	
	sw	$t3	404($a3)		
	mov	$t1	$t2		
	addi	$a3	$a3	4	
	b	For0			
endFor0:					

Naïve ILP Implementation

- It would be “nice” if we could use the 4 ALU & 2 I/O unit to code the loop in a format such as:

  mul $t1, $t1, $a0|| mul $t2, $t2, $a1 || add $a3, $t1, $t2 ||
  mov $t1, $t2 || lw $t1, 0($a3)|| sw $t1, 404($a3)||

- But this have several source-destination hazards.
- Hence we have to introduce a software pipeline

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”

Loop Unrolling / Software Pipelining

- Loop unrolling – multiple copies of the loop body are made and instructions from different iterations are scheduled together as a way to increase ILP

- Apply loop unrolling (4 times for our example) and then schedule the resulting code
  - Eliminate unnecessary loop overhead instructions
  - Schedule so as to avoid hazards (pipeline)

- During unrolling the compiler can apply register renaming to eliminate data dependencies that are not true data dependencies

ILP Software Pipeline with Pseudo Code

Let x_i, y_i stand for x[n+i], y[n+i]

Prolog:					
Load x0	Load x1				i=0
Multiply x0*a1	Multiply x1*a0	Load x3			i++
Add (y1)	Multiply x2*a0	Multiply x3*a1	Load x4		i++
Loop:					
Add (y2)	Multiply x3*a0	Multiply x4*a1	Load x5	Store y1	i++
Add (y3)	Multiply x4*a0	Multiply x5*a1	Load x6	Store y2	i++
Add (y4)	Multiply x5*a0	Multiply x6*a1	Load x7	Store y3	i++
Add (y5)	Multiply x6*a0	Multiply x7*a1	Load x8	Store y4	i++
Epilog:					
Add (y6)	Multiply x7*a0	Multiply x8*a1	Store y5		i++
Add (y7)	Store y6				i++
Store y7					i++

MIPS Code

ALU-0	ALU-1	ALU-2	ALU-3	LW/SW-0	LW/SW-1	
Init:						
	la $a3, x	li $a0, 3	li $a1, 2			
	addi, $a2,$a3,384	sw, $0, 400($a3)				
Prolog:						
				lw $t0, 0($a3)	lw $t1, 4($a3)	x[0], x[1]
	mul $v0, $t0, $a1	mul $v1, $t1, $a0		lw $t2, 8($a3)		x[2]
add $s1,$v0,$v1	mul $s8, $t1, $a1	mul $t8, $t2, $a0		lw $t3, 12($a3)		x[3]
						n=1
Loop:	bge $a2, $a3, Epilog					
add $s2, $s8, $t8	mul $v0, $t2, $a1	mul $v1, $t3, $a0		lw $t4, 16($a3)	sw $s1, 404($a3)	x[n+3], y[n]
add $s3, $v0, $v1	mul $s8, $t3, $a1	mul $t8, $t4, $a0		lw $t5, 20($a3)	sw $s2, 408($a3)	x[n+4], y[n+1]
add $s4, $s8, $t8	mul $v0, $t4, $a1	mul $v1, $t5, $a0		lw $t6, 24($a3)	sw $s3, 412($a3)	x[n+5], y[n+2]
add $s5, $v0, $v1	mul $s8, $t5, $a1	mul $t8, $t6, $a0		lw $t7, 28($a3)	sw $s4, 416($a3)	x[n+6], y[n+3]
	b Loop		addi $a3, $a3, 16			
Epilog:						
add $s6, $s8, $t8	mul $v0, $t6, $a1	mul $v1, $t7, $a0			sw $s5, 404($a3)	y[97]

Performance Analysis (of the example)

- About 6x duplication of certain units (which occupy relatively small portion of the die)
- There is a 5x speedup in the loop.
- Generally, there would be some overhead inside loops including overhead for prologue, epilogue, and loop handling
  - hence a speedup of 3.6-3.8 for 4 ALUs is an excellent achievement.
- Generally, there would be some overhead outside loops and the program might include parts that cannot be parallelized
  - Hence speedup is limited according to Amdahl's law
- Identifying and utilizing instruction level ILP by the compiler and/ or the programmer is not trivial
  - Will be discussed later

Intermediate Summary

- Idea
  - Duplicate compute Units ➔ more than one pipeline
  - Fetch numerous instructions simultaneously
  - Utilize multiple issues (more than one stream of instructions)
  - Consider a set of available (fetched) instructions
  - Identify independent instructions
  - Execute independent instructions
- How to (who does) identify independent instructions?

ILP in More Depth

- The ILP of a program is a theoretical measure of the average number of program's instructions that can be execute simultaneously
- Determined by the number of data and control dependencies
- ILP is generally applied to a single instruction stream with single data stream
  - Nevertheless, in a multi threading environment dependencies tend to decrease. Hence, ILP enhanced machine can be very efficient WRT thread level parallelism

Extracting Yet *More* Performance

- Fetch (and execute) more than one instructions at one time (expand every pipeline stage to accommodate multiple instructions) – multiple-issue
- How many instructions to issue in one clock cycle – issue slots

Static vs. Dynamic ILP

- One issue to be addressed is "who" is responsible for identifying ILP and which instructions to execute simultaneously
- In static ILP exploiting environments the decisions are done prior to the execution by the assembly language programmer and / or by the compiler
  - For example loop unrolling
  - Examples include StarCore SC140, Intel Itanium and the IA-64.
- In dynamic ILP exploiting environments (also referred to as super-scalars) identifying and exploiting ILP is done by the hardware at run time
  - Example includes IBM Power 2, Pentium Pro, and MIPS R10K

In order vs. Out of Order Issue/Commit

- To better exploit ILP; instructions might be executed in an “out-of-order” fashion
- This means that committing the results of instructions might be done out of order

Summary and Conclusions (ILP)

- A method for exploiting parallelism at the data level has been presented and analyzed
- An example using MIPS ISA and an imaginary SIMD extension to MIPS is used
- Speedup has been briefly analyzed