



**POLITECNICO**  
MILANO 1863

DIPARTIMENTO DI ELETTRONICA  
INFORMAZIONE E BIOINGEGNERIA

POLITECNICO MILANO 1863  
**NECST**  
laboratory

# Exercise Session 7

VLIW, More VLIW, Dynamic Branch Prediction, (Extra: Simple Scheduling)  
Advanced Computer Architectures

12th May 2025

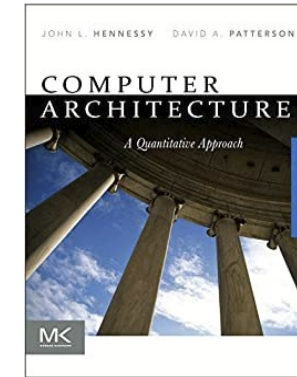
Davide Conficconi <[davide.conficconi@polimi.it](mailto:davide.conficconi@polimi.it)>

## Recall: Material (EVERYTHING OPTIONAL)

<https://webeep.polimi.it/course/view.php?id=14754>

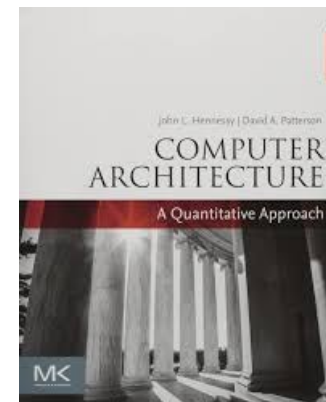
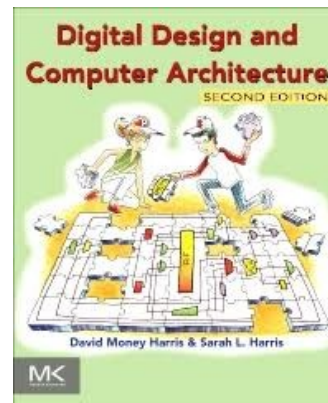
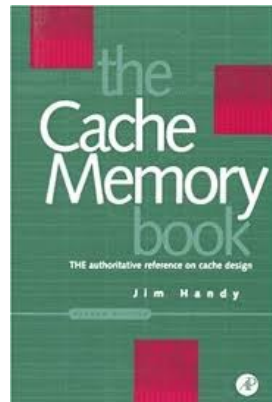
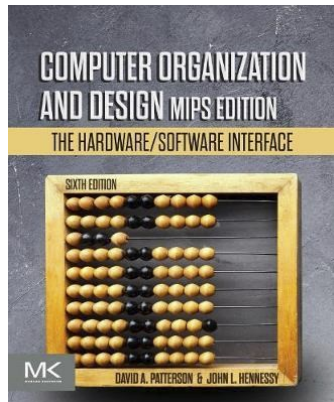
<https://tinyurl.com/aca-grid25>

Textbook: Hennessy and Patterson, Computer Architecture: A Quantitative Approach



Application Software	<code>&gt;"hello world!"</code>
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

### Other Interesting Reference



**SUPERCALIFRAGILIS  
TICHESPIRALIDOSO  
(AKA VLIW)**



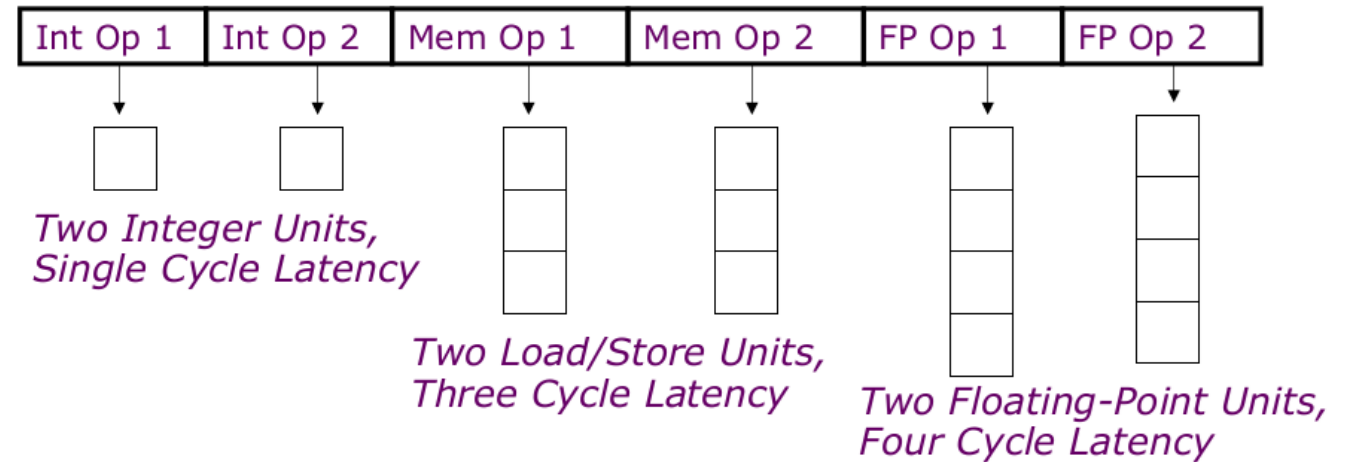
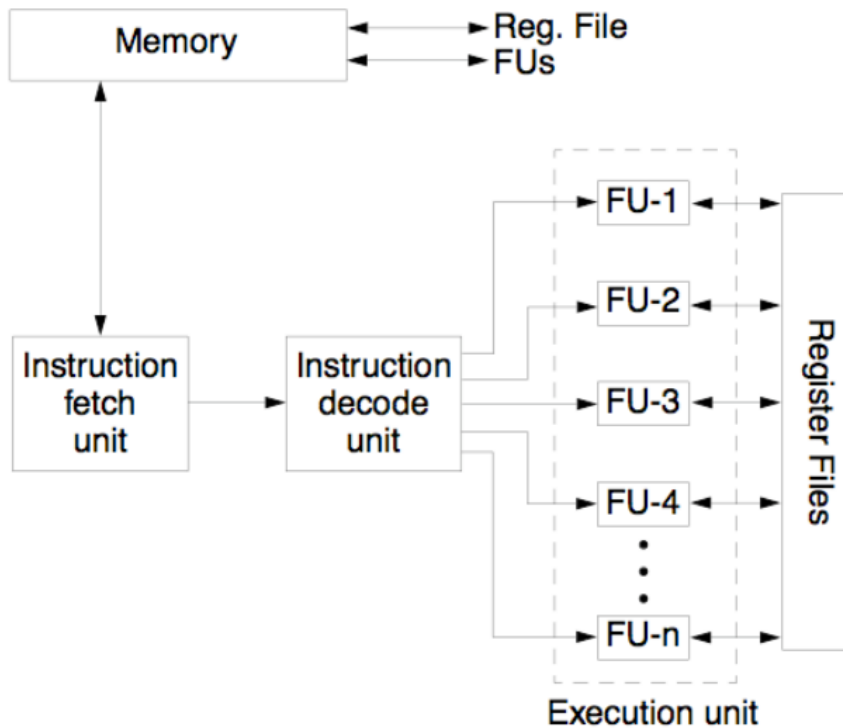
# Recall: Instruction Level Parallelism

Two strategies to support ILP:

- **Dynamic Scheduling:** Depend on the hardware to locate parallelism
- **Static Scheduling:** Rely on software for identifying potential parallelism

Hardware intensive approaches dominate desktop and server markets

# Recall: VLIW and Static Scheduling



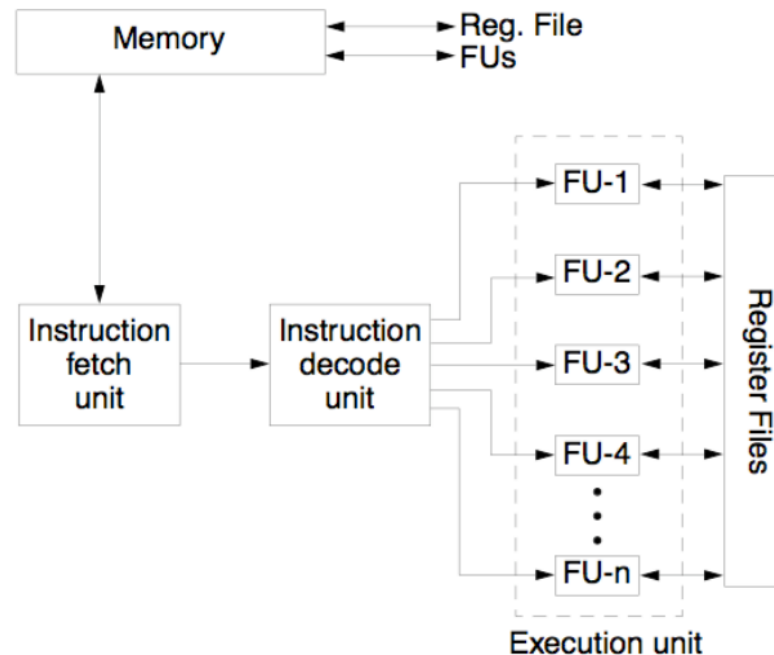
**Static Scheduling:** Rely on software for identifying potential parallelism  
(example of List Based Scheduling with ASAP)

VLIW (Very Long Instruction Word) processors expect dependency-free code.

## Question on VLIW Architecture

A VLIW Architecture has to have multiple Program Counters to load the necessary Multiple Data.  
Given the previous statement, confirm if it is TRUE or FALSE and **effectively support** your answer.

Circle the **right** answer:      True      False





# Recall: VLIW Compiler Responsibilities

## The compiler:

Schedules to maximize parallel execution

Exploit ILP and LLP (Loop Level Parallelism)

It is necessary to map the instructions over the machine functional units

This mapping must account for **time constraints** and dependencies among the tasks

Guarantees intra-instruction parallelism

Schedules to **avoid data hazards** (no interlocks)

Typically separates operations with **explicit NOPs**

The goal is to minimize the total execution time for the program

VLIW processors expect **dependency-free code**.

# Recall: Static Scheduling: methods

- Simple code motion
- Loop unrolling & loop peeling
- Software pipeline
- Global code scheduling (across basic block)
  - Trace scheduling
  - Superblock scheduling
  - Hyperblock scheduling
  - Speculative Trace scheduling



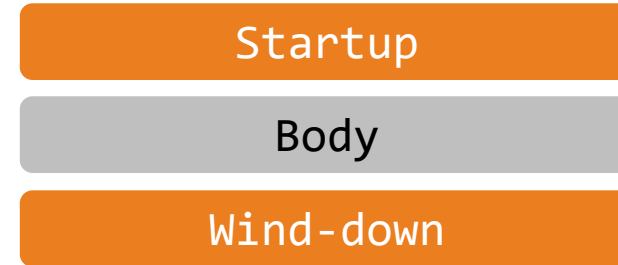
# Recall: The Situation

Program memory



Compiled code

**loop:**

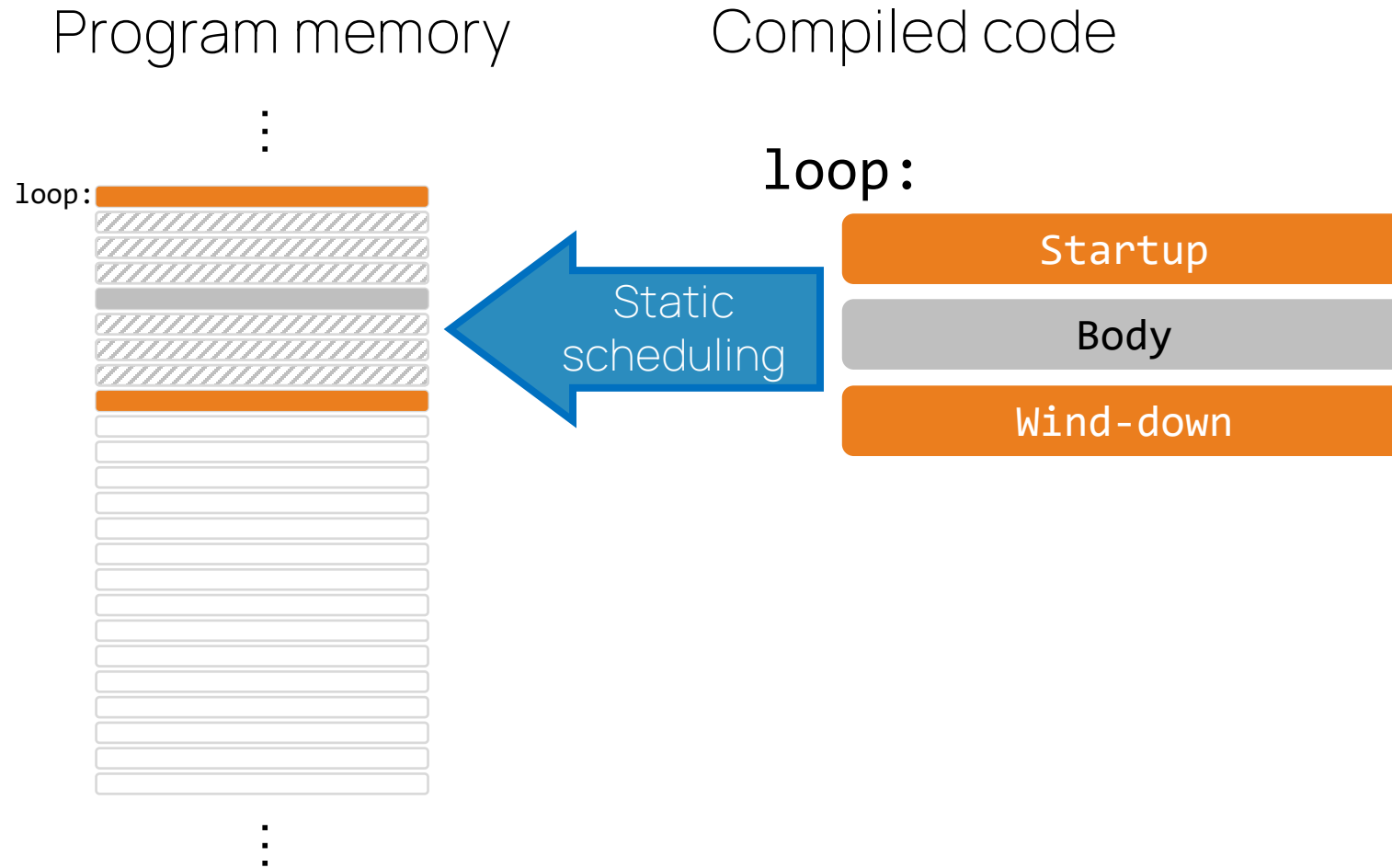


e.g. loads

e.g. floating-point operations

e.g. stores

# Recall: Static Scheduling



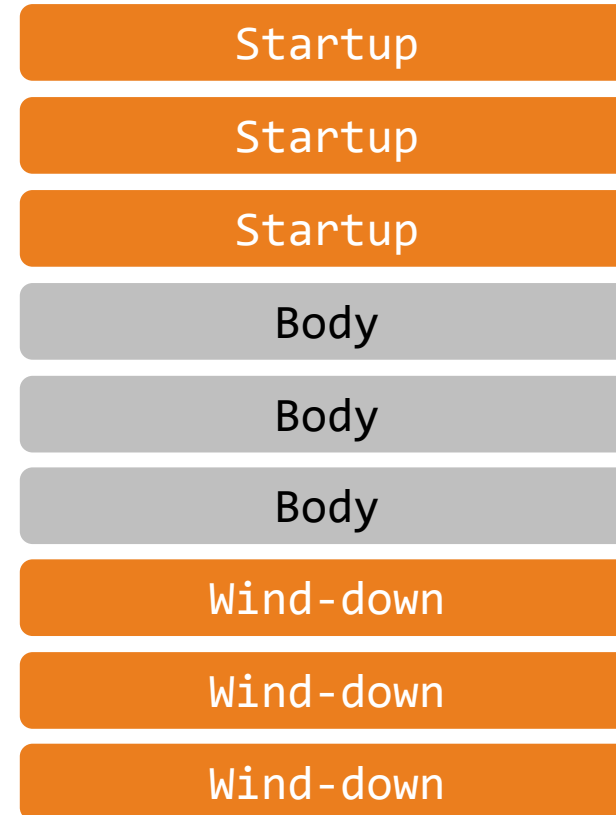
# Recall: Static Scheduling → Loop Unrolling

Program memory

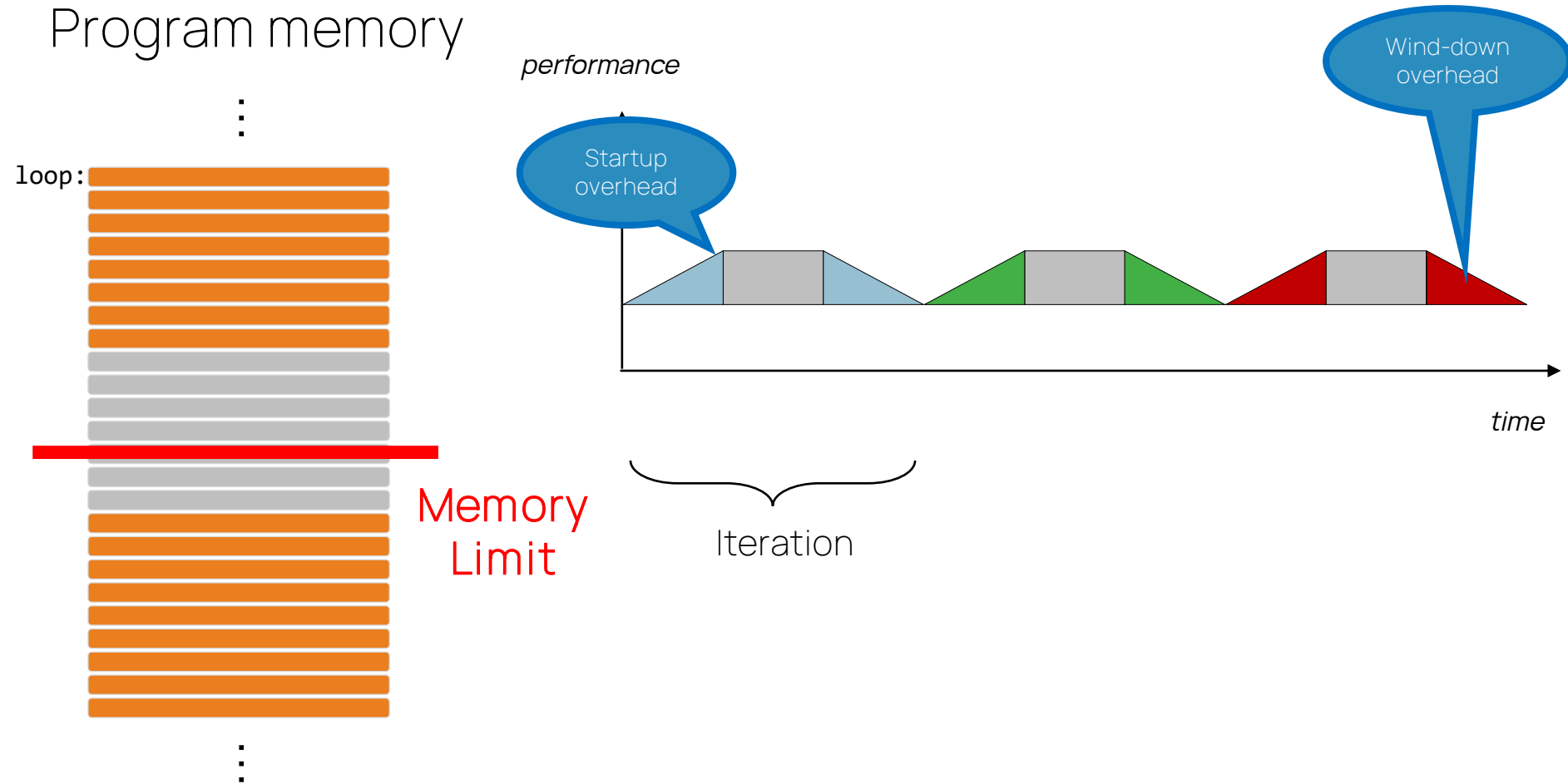


Compiled code

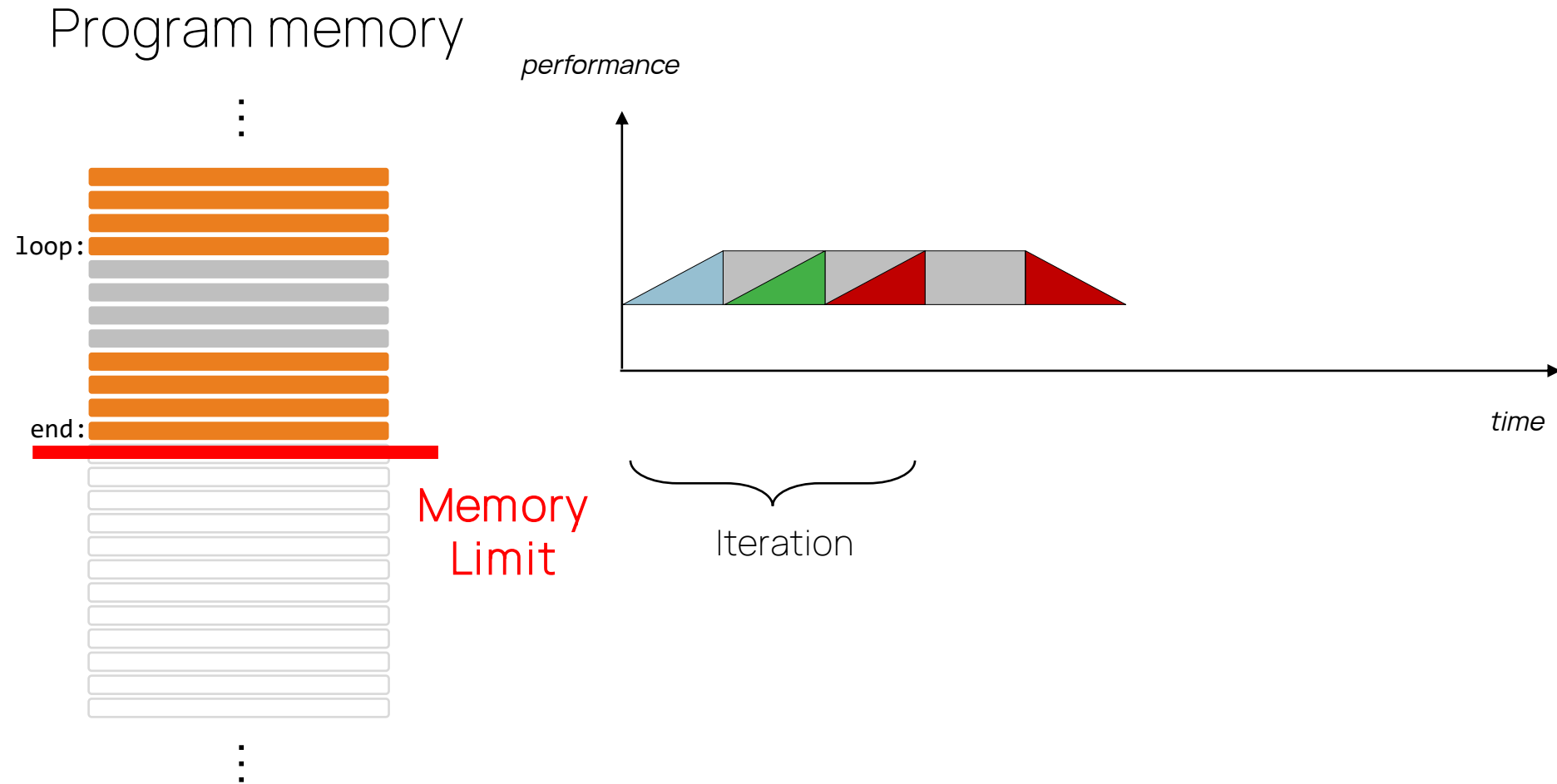
loop:



# Recall: Static Scheduling → Code Explosion



# Recall: Static Scheduling → Software Pipelining



## Exe VLIW: Architecture

- Consider the program be executed on a **3-issue VLIW MIPS** (Very Long Instruction Word) architecture with **3 non-pipelined functional** units
- **Integer ALU** with **1** cycle latency to **next Integer/FP** and **2** cycle latency to **next Branch**
- **Memory** Unit with **3** cycle latency
- **Floating Point** Unit with **3** cycle latency
- Branch completed with **1 cycle delay slot** (branch solved in ID stage)

## Exe VLIW: Schedule

- Considering **one iteration** of the loop
- schedule the assembly code for the 3-issue VLIW machine in the following table by using the **list-based scheduling** with **ASAP**
- **Do not** use neither software pipelining nor loop unrolling nor modifying loop indexes.
- Please do not need to write in NOPs (can leave blank).



## Exe VLIW: the code

Assembly Code:

```
L1: lw    $2, A($4)  
    addi  $2, $2, 4  
    lw     $3, B($4)  
    addi  $6, $2, -5  
    sub   $5, $4, $3  
    sw    $5, C($4)  
    addi  $4, $4, 4  
    bne   $4, $7, L1
```

## Exe VLIW: schedule

**L1:** **lw** \$2, A(\$4)  
**addi** \$2, \$2, 4  
**lw** \$3, B(\$4)  
**addi** \$6, \$2, -5  
**sub** \$5, \$4, \$3  
**sw** \$5, C(\$4)  
**addi** \$4, \$4, 4  
**bne** \$4, \$7, L1

**ALU** 1 cc Integer, 2 cc Branch

**MU** 3 cc

**FPU** 3 cc

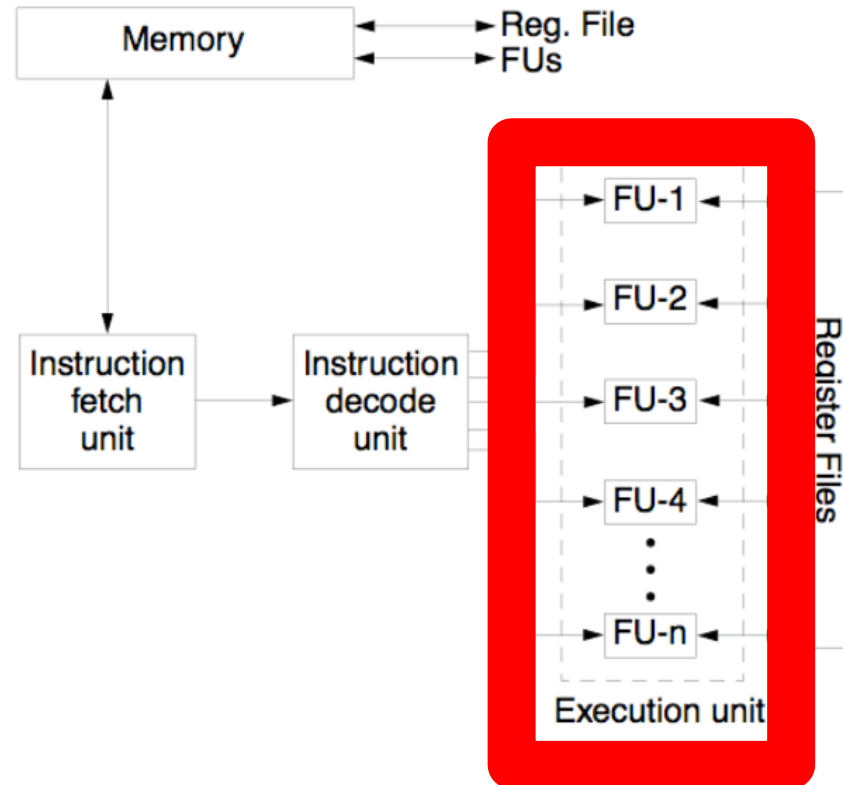
# Exe VLIW: schedule

**L1:** lw \$2, A(\$4)  
addi \$2, \$2, 4  
lw \$3, B(\$4)  
addi \$6, \$2, -5  
sub \$5, \$4, \$3  
sw \$5, C(\$4)  
addi \$4, \$4, 4  
bne \$4, \$7, L1


**ALU** 1 cc Integer, 2 cc Branch

**MU** 3 cc

**FPU** 3 cc



# Exe VLIW: schedule


**lw** \$2, A(\$4)  
**addi** \$2, \$2, 4  
**lw** \$3, B(\$4)  
**addi** \$6, \$2, -5  
**sub** \$5, \$4, \$3  
**sw** \$5, C(\$4)  
**addi** \$4, \$4, 4  
**bne** \$4, \$7, L1

	Integer ALU(1/2 b)	Memory Unit(3cc)	FPU(3cc)
C1			
C2			
C3			
C4			
C5			
C6			
C7			
C8			
C9			
C10			
C11			
C12			
C13			
C14			
C15			

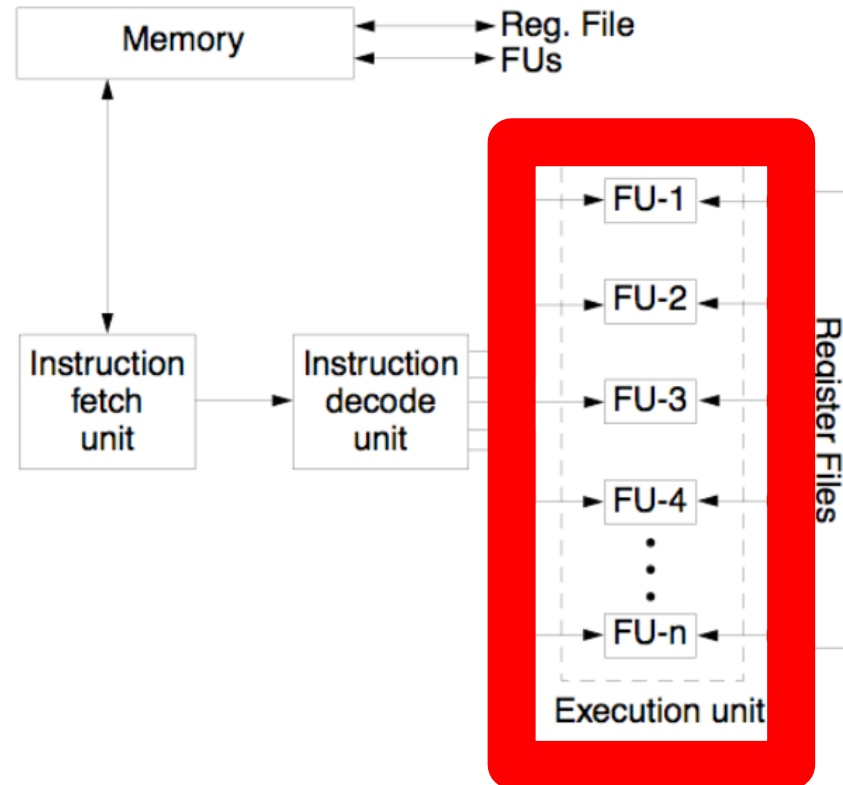
## Exe 1.2 VLIW: schedule PIPELINED

L1: lw \$2, A(\$4)  
addi \$2, \$2, 4  
lw \$3, B(\$4)  
addi \$6, \$2, -5  
sub \$5, \$4, \$3  
sw \$5, C(\$4)  
addi \$4, \$4, 4  
bne \$4, \$7, L1


**ALU** 1 cc Integer, 2 cc Branch

**MU** 3 cc PIPELINED

**FPU** 3 cc PIPELINED




# Exe 1 VLIW: schedule PIPELINED


**lw** \$2, A(\$4)  
**addi** \$2, \$2, 4  
**lw** \$3, B(\$4)  
**addi** \$6, \$2, -5  
**sub** \$5, \$4, \$3  
**sw** \$5, C(\$4)  
**addi** \$4, \$4, 4  
**bne** \$4, \$7, L1

	Integer ALU(1/2 b)	Memory Unit(3cc)	FPU(3cc)
C1			
C2			
C3			
C4			
C5			
C6			
C7			
C8			
C9			
C10			
C11			
C12			
C13			
C14			
C15			

# Exe 1 VLIW: schedule PIPELINED FLOPS/CC

 `lw $2, A($4)`  
`addi $2, $2, 4`  
`lw $3, B($4)`  
`addi $6, $2, -5`  
`sub $5, $4, $3`  
`sw $5, C($4)`  
`addi $4, $4, 4`  
`bne $4, $7, L1`

FLOPs/CC=

	Integer ALU(1/2 b)	Memory Unit(3cc)	FPU(3cc)
C1			
C2			
C3			
C4			
C5			
C6			
C7			
C8			
C9			
C10			
C11			
C12			
C13			
C14			
C15			



# Exe 1 VLIW: Pipeline Scheduling

	Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14
1	NOP   NOP   NOP														
2	NOP   NOP   NOP														
3	NOP   NOP   NOP														
4	NOP   NOP   NOP														
5	NOP   NOP   NOP														
6	NOP   NOP   NOP														
7	NOP   NOP   NOP														
8	NOP   NOP   NOP														
9	NOP   NOP   NOP														

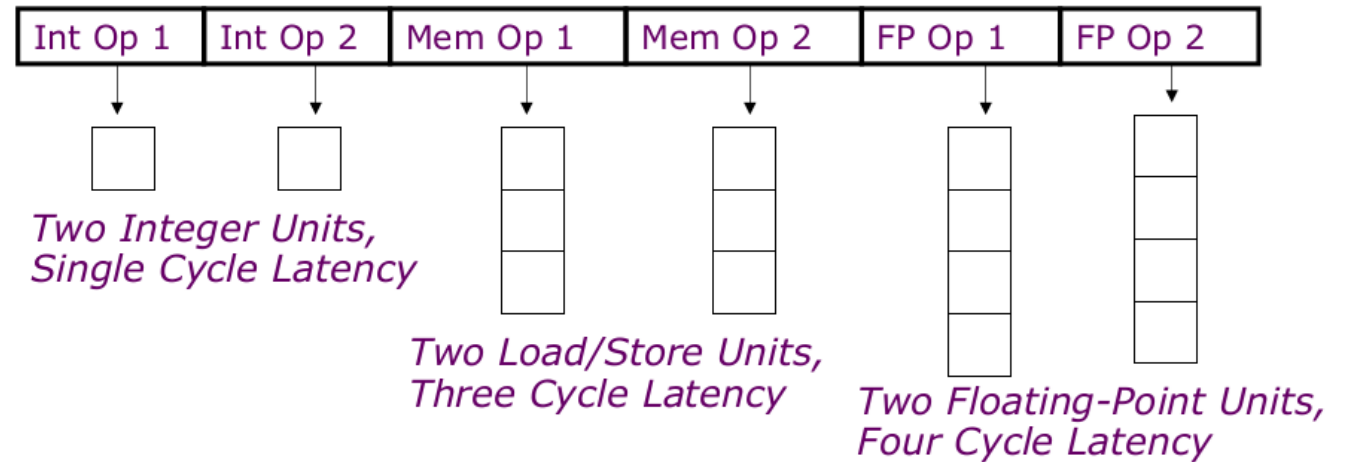
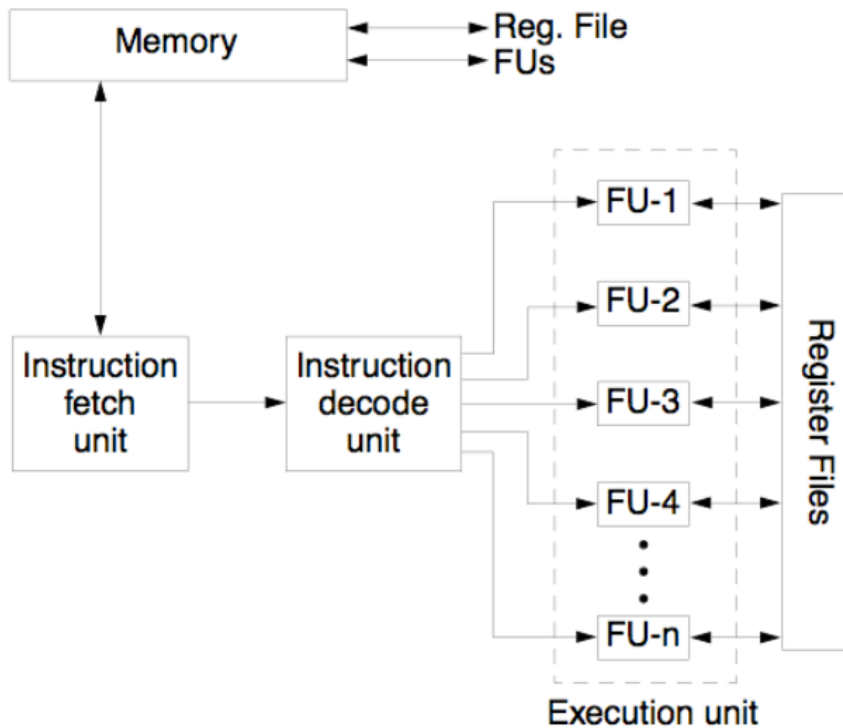
**ALU** 1 cc Integer, 2 cc Branch, **MU** 3 cc PIPELINED, **FPU** 3 cc PIPELINED







# Recall: VLIW and Static Scheduling



**Static Scheduling:** Rely on software for identifying potential parallelism  
(example of List Based Scheduling with ASAP)

VLIW (Very Long Instruction Word) processors expect dependency-free code.

## Exe 1 VLIW: Architecture

- Consider the program be executed on a **3-issue VLIW MIPS** (Very Long Instruction Word) architecture with **3 fully pipelined functional** units
- **Integer ALU** with **1** cycle latency
- **Memory Unit** with **2** cycle latency
- **Floating Point Unit** with **3** cycle latency
- Branch solved in EXE stage, **no early evaluation**

## Exe VLIW.1: schedule

- Considering **one iteration** of the loop
- schedule the assembly code for the 3-issue VLIW machine in the following table by using the **list-based scheduling** with **ASAP**
- Calculate the **performance** (FLOPs per cycle)
- **Do not** use neither software pipelining nor loop unrolling nor modifying loop indexes
- Please do not need to write in NOPs (can leave blank)

## Exe VLIW.2: schedule with unroll

- **Unroll** the loop by one iteration (so two iterations of the original loop are performed for every branch in the new assembly code)
- You only need to worry about the steady-state code in the core of the loop (no epilogue or prologue)
- schedule the assembly code for the 3-issue VLIW machine in the following table by using the **list-based scheduling** with **ASAP**
- Calculate the **performance** (FLOPs per cycle)
  
- **Do not** use software pipelining
- Please do not need to write in NOPs (can leave blank)

## Exe VLIW.1: the code

C Code:

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

Assembly Code:



## Exe VLIW.1: the code

C Code:

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

Assembly Code:

```
loop: ld      f1, 0(r1)  
      ld      f2, 0(r2)  
      fmul    f1, f1, f1  
      fadd    f1, f1, f2  
      st      f1, 0(r3)  
      addi    r1, r1, 4  
      addi    r2, r2, 4  
      addi    r3, r3, 4  
      bne     r3, r4, loop
```

## Exe VLIW.1: schedule

<b>loop:</b>	<b>ld</b>	<b>f1, 0(r1)</b>	<b>ALU</b>	<b>1 cc</b>
	<b>ld</b>	<b>f2, 0(r2)</b>	<b>MU</b>	<b>2 cc</b>
	<b>fmul</b>	<b>f1, f1, f1</b>	<b>FPU</b>	<b>3 cc</b>
	<b>fadd</b>	<b>f1, f1, f2</b>		
	<b>st</b>	<b>f1, 0(r3)</b>		
	<b>addi</b>	<b>r1, r1, 4</b>		
	<b>addi</b>	<b>r2, r2, 4</b>		
	<b>addi</b>	<b>r3, r3, 4</b>		
	<b>bne</b>	<b>r3, r4, loop</b>		

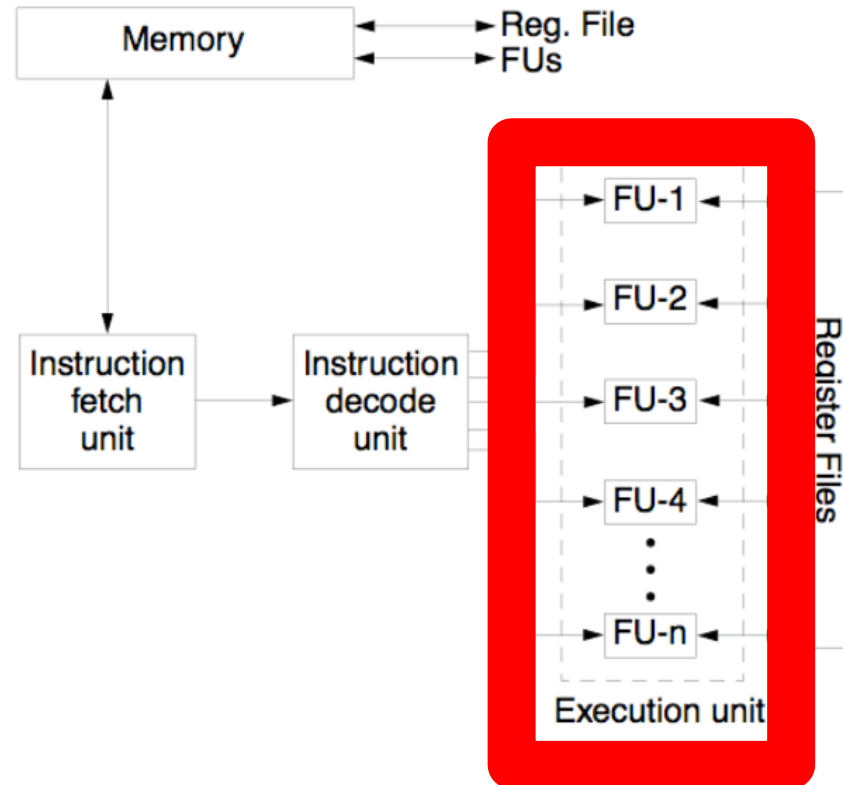
## Exe VLIW.1: schedule

```
loop: ld    f1, 0(r1)
      ld    f2, 0(r2)
      fmul  f1, f1, f1
      fadd  f1, f1, f2
      st    f1, 0(r3)
      addi  r1, r1, 4
      addi  r2, r2, 4
      addi  r3, r3, 4
      bne   r3, r4, loop
```


**ALU** 1 cc

**MU** 2 cc

**FPU** 3 cc



## Exe VLIW.1: schedule

: **ld f1, 0(r1)**  
**ld f2, 0(r2)**  
**fmul f1, f1, f1**  
**fadd f1, f1, f2**  
**st f1, 0(r3)**  
**addi r1, r1, 4**  
**addi r2, r2, 4**  
**addi r3, r3, 4**  
**bne r3, r4, loop**

	Integer ALU (1 cc)	Memory Unit (2 cc)	FPU (3 cc)
C1			
C2			
C3			
C4			
C5			
C6			
C7			
C8			
C9			
C10			
C11			
C12			
C13			
C14			
C15			

## Exe VLIW.2: schedule with unroll

- **Unroll** the loop by one iteration (so two iterations of the original loop are performed for every branch in the new assembly code)
- You only need to worry about the steady-state code in the core of the loop (no epilogue or prologue)
- schedule the assembly code for the 3-issue VLIW machine in the following table by using the **list-based scheduling** with **ASAP**
- Calculate the **performance** (FLOPs per cycle)
  
- **Do not** use software pipelining
- Please do not need to write in NOPs (can leave blank)

## Exe VLIW.2: the code

C Code:

```
for(int i=0; i<N; i+=2) {  
    C[i] = A[i]*A[i] + B[i];  
    C[i+1] = A[i+1]*A[i+1] + B[i+1];  
}
```

Assembly Code:

## Exe VLIW.2: the code

C Code:

```
for(int i=0; i<N; i+=2) {  
    C[i] = A[i]*A[i] + B[i];  
    C[i+1] = A[i+1]*A[i+1] + B[i+1];  
}
```

Assembly Code:

```
loop: ld      f1, 0(r1)  
      ld      f3, 4(r1)  
      ld      f2, 0(r2)  
      ld      f4, 4(r2)  
      fmul    f1, f1, f1  
      fmul    f3, f3, f3  
      fadd    f1, f1, f2  
      fadd    f3, f3, f4  
      st      f1, 0(r3)  
      st      f3, 4(r3)  
      addi    r1, r1, 8  
      addi    r2, r2, 8  
      addi    r3, r3, 8  
      bne     r3, r4, loop
```



## Exe VLIW.2: schedule

<b>loop:</b>	<b>ld</b>	<b>f1, 0(r1)</b>	<b>ALU</b>	<b>1 cc</b>
	<b>ld</b>	<b>f3, 4(r1)</b>	<b>MU</b>	<b>2 cc</b>
	<b>ld</b>	<b>f2, 0(r2)</b>	<b>FPU</b>	<b>3 cc</b>
	<b>ld</b>	<b>f4, 4(r2)</b>		
	<b>fmul</b>	<b>f1, f1, f1</b>		
	<b>fmul</b>	<b>f3, f3, f3</b>		
	<b>fadd</b>	<b>f1, f1, f2</b>		
	<b>fadd</b>	<b>f3, f3, f4</b>		
	<b>st</b>	<b>f1, 0(r3)</b>		
	<b>st</b>	<b>f3, 4(r3)</b>		
	<b>addi</b>	<b>r1, r1, 8</b>		
	<b>addi</b>	<b>r2, r2, 8</b>		
	<b>addi</b>	<b>r3, r3, 8</b>		
	<b>bne</b>	<b>r3, r4, loop</b>		

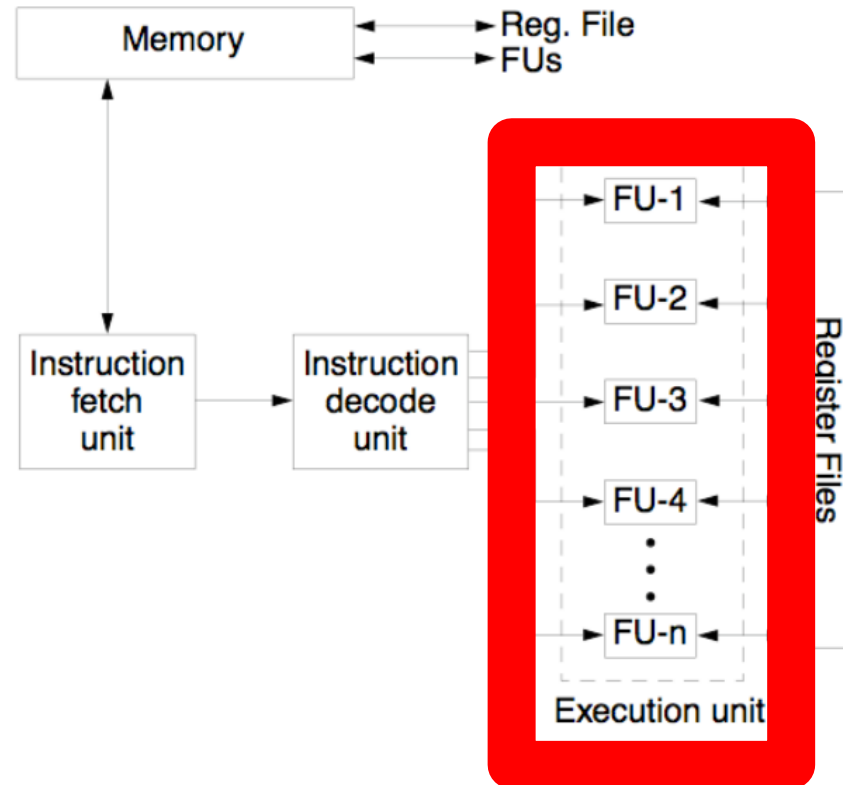
## Exe VLIW.2: schedule

```
loop: ld    f1, 0(r1)
      ld    f3, 4(r1)
      ld    f2, 0(r2)
      ld    f4, 4(r2)
      fmul  f1, f1, f1
      fmul  f3, f3, f3
      fadd  f1, f1, f2
      fadd  f3, f3, f4
      st    f1, 0(r3)
      st    f3, 4(r3)
      addi  r1, r1, 8
      addi  r2, r2, 8
      addi  r3, r3, 8
      bne   r3, r4, loop
```


**ALU** 1 cc

**MU** 2 cc

**FPU** 3 cc



## Exe VLIW.2: schedule


**1** **ld f1, (r1)**  
**ld f3, 4(r1)**  
**ld f2, 0(r2)**  
**ld f4, 4(r2)**  
**fmul f1, f1, f1**  
**fmul f3, f3, f3**  
**fadd f1, f1, f2**  
**fadd f3, f3, f4**  
**st f1, 0(r3)**  
**st f3, 4(r3)**  
**addi r1, r1, 8**  
**addi r2, r2, 8**  
**addi r3, r3, 8**  
**bne r3, r4, loop**

	Integer ALU (1 cc)	Memory Unit (2 cc)	FPU (3 cc)
C1			
C2			
C3			
C4			
C5			
C6			
C7			
C8			
C9			
C10			
C11			
C12			
C13			
C14			
C15			

## Exe VLIW.2: FLOPs/CC

```
loop: ld  f1, (r1)
      ld  f3, 4(r1)
      ld  f2, 0(r2)
      ld  f4, 4(r2)
      fmul f1, f1, f1
      fmul f3, f3, f3
      fadd f1, f1, f2
      fadd f3, f3, f4
      st  f1, 0(r3)
      st  f3, 4(r3)
      addi r1, r1, 8
      addi r2, r2, 8
      addi r3, r3, 8
      bne r3, r4, loop
```



FLOPs/CC=

	Integer ALU (1 cc)	Memory Unit (2 cc)	FPU (3 cc)
C1			
C2			
C3			
C4			
C5			
C6			
C7			
C8			
C9			
C10			
C11			
C12			
C13			
C14			
C15			



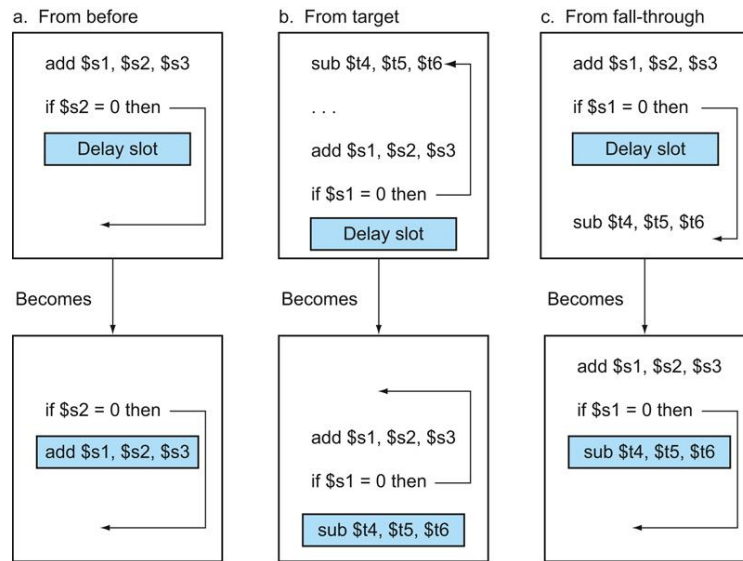




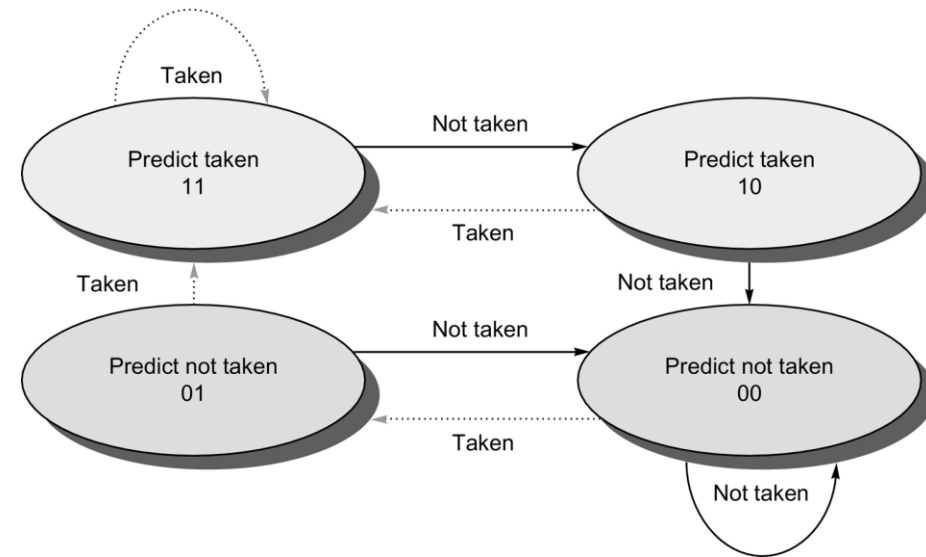
# Prediction

Branch vanguard: decomposing branch functionality into prediction and resolution instructions

The IBM z15 High Frequency Mainframe Branch Predictor



Copyright © 2021 Elsevier Inc. All rights reserved.



# Dynamic Branch Predictor

- Describe (the answer has to be effectively supported) a 1-BHT and a 2-BHT able to execute the following assembly code (R0 is set to 1, R1 is set to 300)

```
      LOOP:  LD  F3 0 (R0)
             ADDD F1 F3 F3
             ADDI R1 R1 3000
LOOP2:  MULTD F2 F2 F3
             SUBI R1 R1 3
             BNEZ R1 LOOP2
             SUBI R0 R0 2
             BNEZ R0 LOOP
```

- The obtained result, in terms of mispredictions, is inline with theoretical characteristics of the two predictors? Please effectively support your answer.

# A First Consideration

```
LOOP:    LD  F3 0 (R0)
          ADDD F1 F3 F3
          ADDI R1 R1 3000
LOOP2:   MULTD F2 F2 F3
          SUBI R1 R1 3
          BNEZ R1 LOOP2
          SUBI R0 R0 2
          BNEZ R0 LOOP
```



# How many iterations?

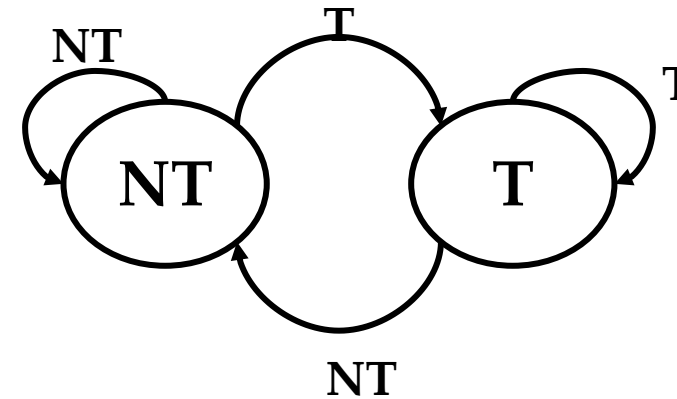
R0 is set to 1  
R1 is set to 300

```
LOOP:    LD  F3  0  (R0)
          ADDD F1  F3  F3
          ADDI R1  R1  3000
LOOP2:   MULTD F2  F2  F3
          SUBI R1  R1  3
          BNEZ R1  LOOP2
          SUBI R0  R0  2
          BNEZ R0  LOOP
```

# 1bit - BHT

```
LOOP:  LD F3 0 (R0)
        ADDD F1 F3 F3
        ADDI R1 R1 3000
LOOP2:  MULTD F2 F2 F3
        SUBI R1 R1 3
        BNEZ R1 LOOP2
        SUBI R0 R0 2
        BNEZ R0 LOOP
```

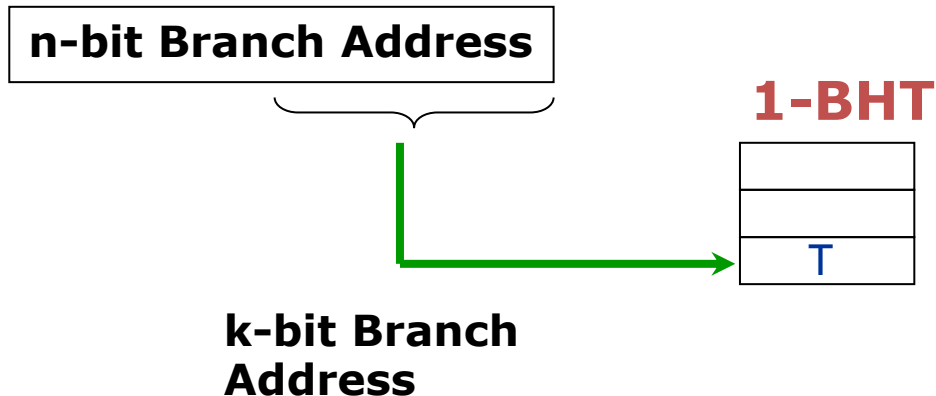
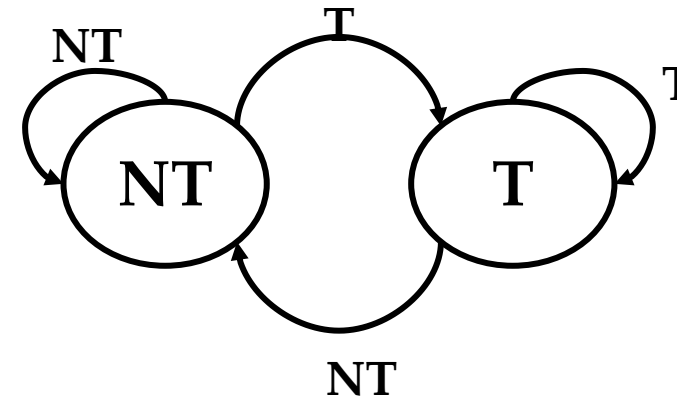
R0 is set to 1  
R1 is set to 300



# 1bit - BHT

```
LOOP:  LD F3 0 (R0)
       ADDD F1 F3 F3
       ADDI R1 R1 3000
LOOP2: MULTD F2 F2 F3
       SUBI R1 R1 3
       BNEZ R1 LOOP2
       SUBI R0 R0 2
       BNEZ R0 LOOP
```

R0 is set to 1  
R1 is set to 300

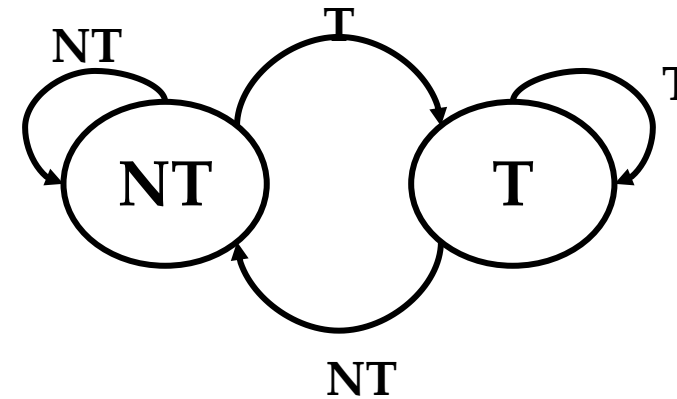


k-bit Branch Address:  
Collide  
Not collide

# 1bit - BHT - Not Collide

```
LOOP:  LD F3 0 (R0)
        ADDD F1 F3 F3
        ADDI R1 R1 3000
LOOP2:  MULTD F2 F2 F3
        SUBI R1 R1 3
        BNEZ R1 LOOP2
        SUBI R0 R0 2
        BNEZ R0 LOOP
```

R0 is set to 1  
R1 is set to 300



Let us consider that the branch addresses do not collide

	1-BHT	1-BHT	1-BHT	1-BHT
LOOP:				
	T	T	NT	NT
LOOP2:	T	NT	T	NT

## 2bit - BHT

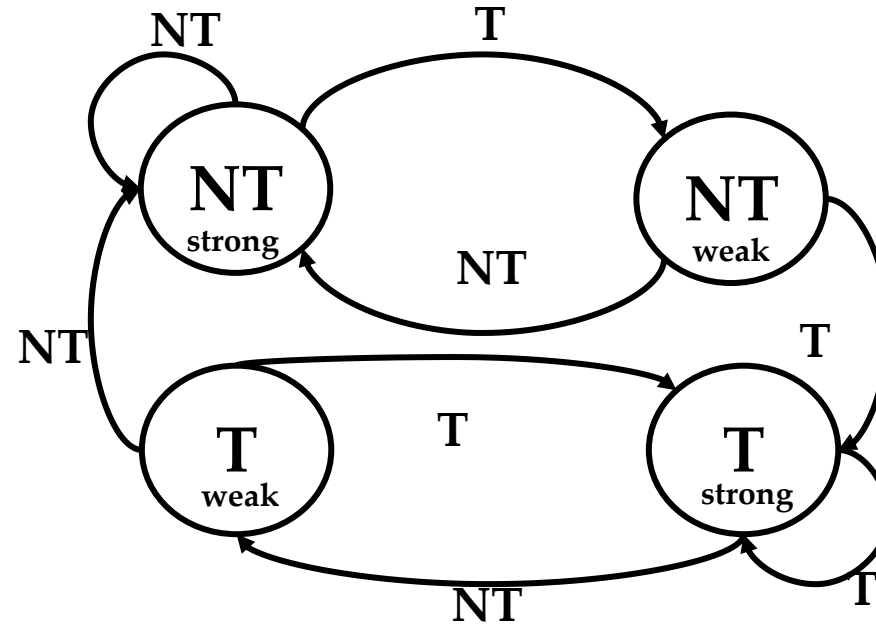
```
LOOP:  LD F3 0 (R0)
        ADDD F1 F3 F3
        ADDI R1 R1 3000
LOOP2: MULTD F2 F2 F3
        SUBI R1 R1 3
        BNEZ R1 LOOP2
        SUBI R0 R0 2
        BNEZ R0 LOOP
```

R0 is set to 1  
R1 is set to 300

## 2bit - BHT

LOOP: LD F3 0 (R0)  
      ADDD F1 F3 F3  
      ADDI R1 R1 3000  
LOOP2: MULTD F2 F2 F3  
      SUBI R1 R1 3  
      BNEZ R1 LOOP2  
      SUBI R0 R0 2  
      BNEZ R0 LOOP

R0 is set to 1  
R1 is set to 300



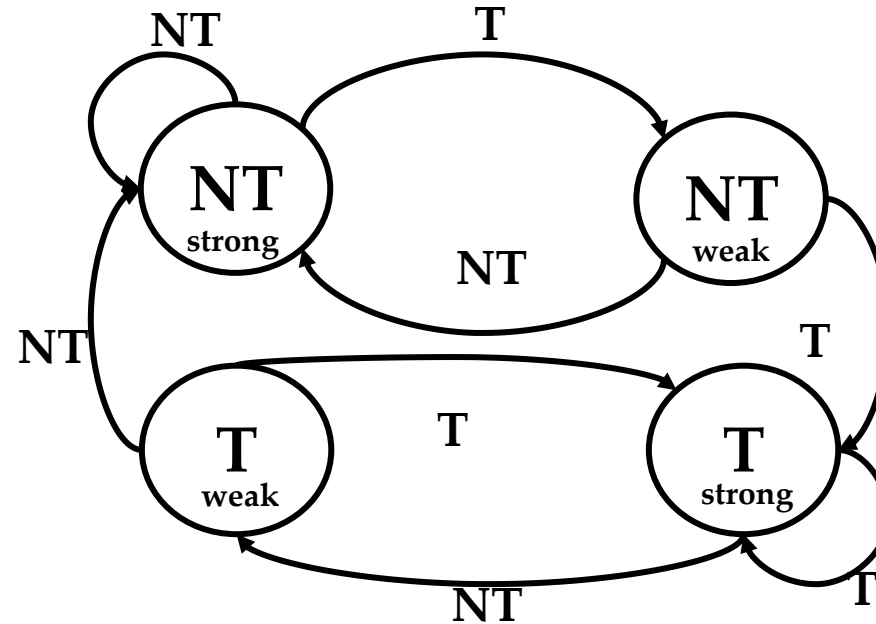
## 2bit - BHT

LOOP: LD F3 0 (R0)  
 ADDD F1 F3 F3  
 ADDI R1 R1 3000

LOOP2: MULTD F2 F2 F3  
 SUBI R1 R1 3  
 BNEZ R1 LOOP2  
 SUBI R0 R0 2  
 BNEZ R0 LOOP

Let us consider  
 that the branch  
 addresses do NOT  
 collide

R0 is set to 1  
 R1 is set to 300



### 2-BHT

LOOP:	NT_strong
LOOP2:	NT_strong

### 2-BHT

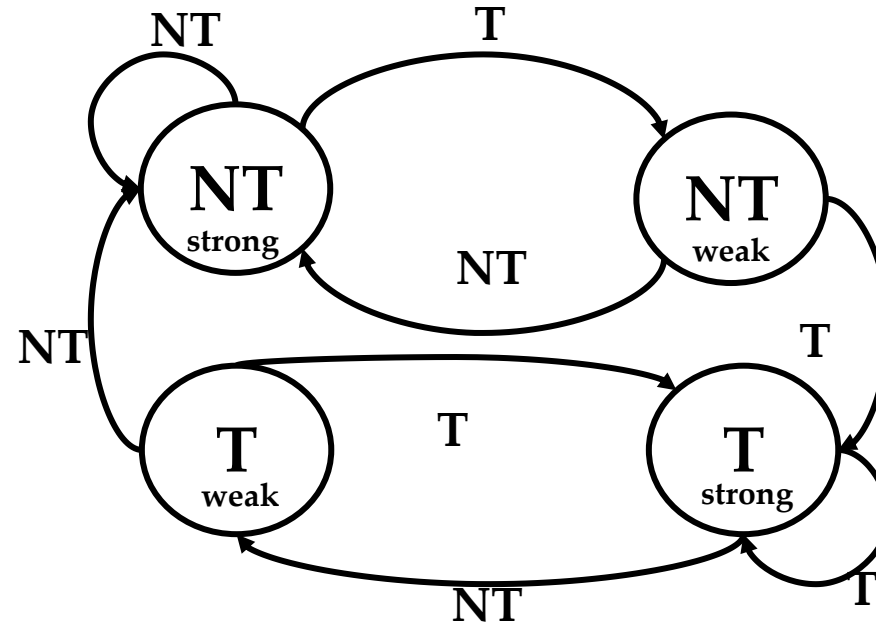
LOOP:	NT_weak
LOOP2:	NT_weak

## 2bit - BHT

**LOOP:** LD F3 0 (R0)  
 ADDD F1 F3 F3  
 ADDI R1 R1 3000  
**LOOP2:** MULTD F2 F2 F3  
 SUBI R1 R1 3  
 BNEZ R1 LOOP2  
 SUBI R0 R0 2  
 BNEZ R0 LOOP

Let us consider  
that the branch  
addresses do NOT  
collide

R0 is set to 1  
 R1 is set to 300



### 2-BHT

LOOP:	
LOOP2:	T <sub>weak</sub>
	T <sub>weak</sub>

### 2-BHT

LOOP:	
LOOP2:	T <sub>strong</sub>
	T <sub>strong</sub>



SUMMARY

Assumption: NO collision

WORST CASES

BEST CASES









# Thanks for your attention

Davide Conficconi <[davide.conficconi@polimi.it](mailto:davide.conficconi@polimi.it)>

## Acknowledgements

E. Del Sozzo, Marco D. Santambrogio, D. Sciuto

Part of this material comes from:

- Paolo Galfano for the static scheduling
- “Computer Organization and Design” and “Computer Architecture A Quantitative Approach” Patterson and Hennessy books
- “Digital Design and Computer Architecture” Harris and Harris
- Elsevier Inc. online materials
- Papers/news cited in this lecture

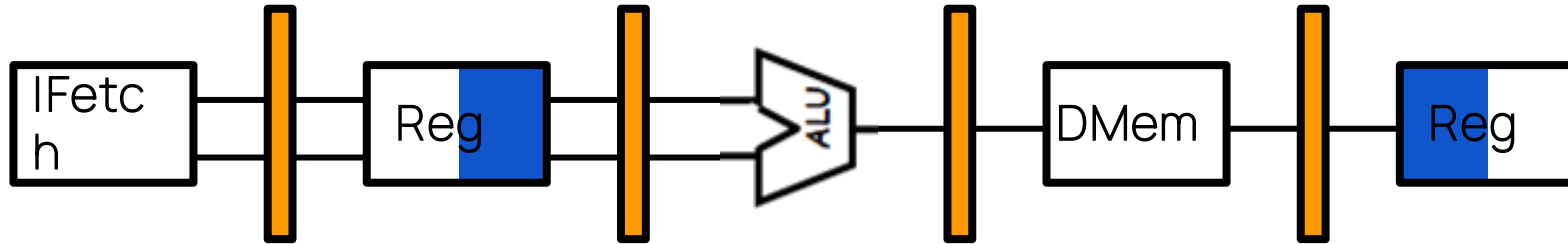
and are **properties of their respective owners**

# Exe 3: Simple Pipelining

## Exe 3 Simple Pipelining : the Code

```
I1:  addi $s3, $s2, 2
I2:  add  $s5, $s4, $s3
I3:  sw   $s5, 4($s3)
I4:  sub  $s7, $s5, $s6
I5:  lw   $s6, 4($s7)
```

## Exe 3: Simple Pipelining: the Architecture



## Exe 3.1 Simple Pipelining : Conflicts

	Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I1:	addi \$s3, \$s2, 2	F	D	E	M	W										
I2:	add \$s5, \$s4, \$s3		F	D	E	M	W									
I3:	sw \$s5, 4(\$s3)			F	D	E	M	W								
I4:	sub \$s7, \$s5, \$s6				F	D	E	M	W							
I5:	lw \$s6, 4(\$s7)					F	D	E	M	W						

Draw the pipeline schema showing all the conflicts/dependencies.

Solve the resulting RAW hazards without using rescheduling and path forwarding.

## Exe 3.1 Simple Pipelining : solve as is

Istr	CK1	CK2	CK3	CK4	CK5	CK6	CK7	CK8	CK9	CK10	CK11
I1											
I2											
I3											
I4											
I5											
Istr	CK12	CK13	CK14	CK15	CK16	CK17	CK18	CK19	CK20	CK21	CK22
I1											
I2											
I3											
I4											
I5											

I1: addi \$s3, \$s2, 2  
 I2: sub \$s4, \$s3, \$s1  
 I3: add \$s5, \$s4, \$s1  
 I4: lw \$s6, 4(\$s4)  
 I5: sub \$s7, \$s4, \$s6



## Exe 3.2 Simple Pipelining : Rescheduling

Reschedule the instructions to **reduce the stalls**; Draw the pipeline schema showing all the data conflicts/dependencies.

## Exe 3.3 Simple Pipelining : FWD Paths

Istr	CK1	CK2	CK3	CK4	CK5	CK6	CK7	CK8	CK9	CK10	CK11
I1											
I2											
I3											
I4											
I5											
Istr	CK12	CK13	CK14	CK15	CK16	CK17	CK18	CK19	CK20	CK21	CK22
I1											
I2											
I3											
I4											
I5											

I1: addi \$s3, \$s2, 2  
 I2: sub \$s4, \$s3, \$s1  
 I3: add \$s5, \$s4, \$s1  
 I4: lw \$s6, 4(\$s4)  
 I5: sub \$s7, \$s4, \$s6