

Static Scheduling and Very Long Instruction Words (VLIW)

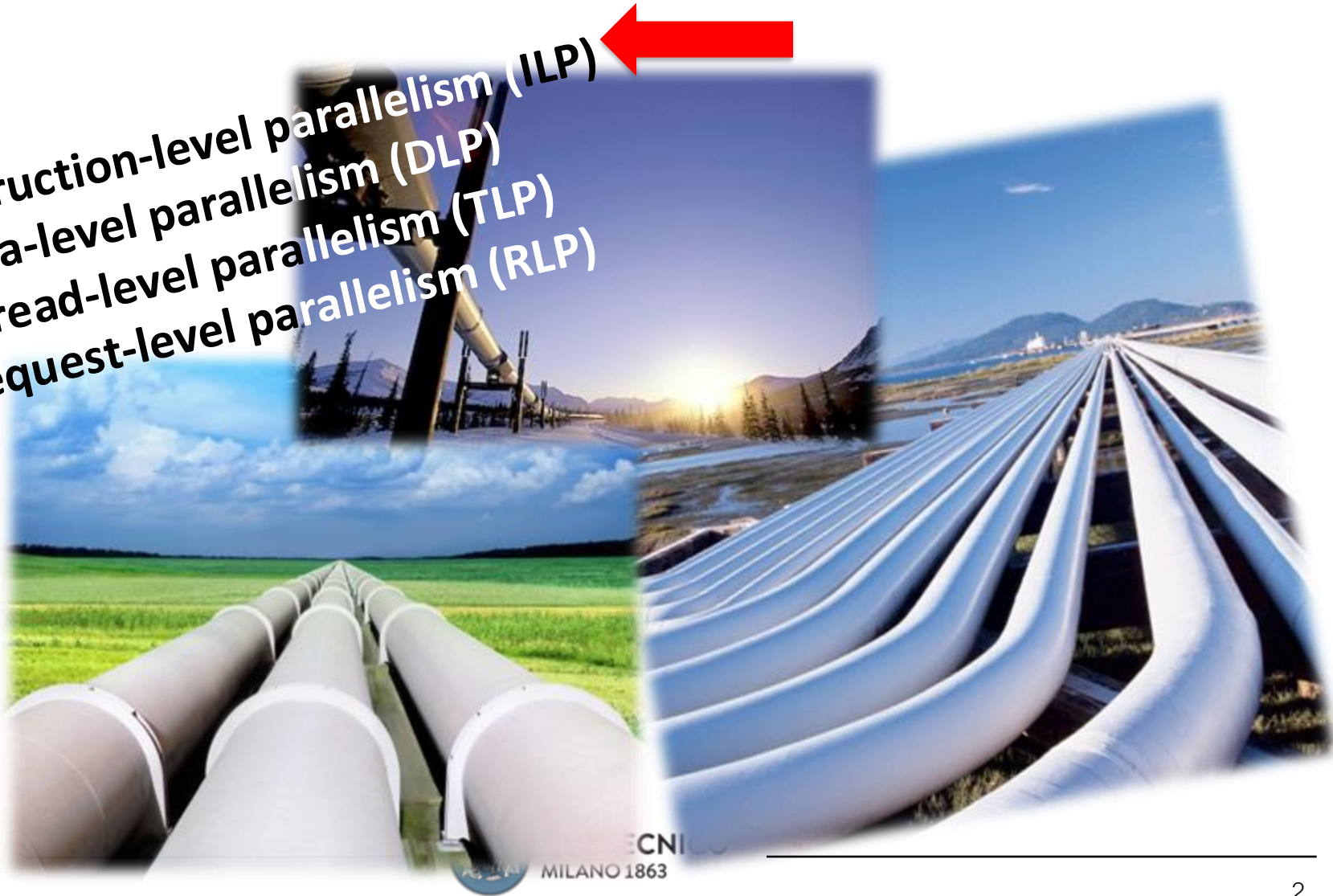
Politecnico di Milano

v2

Christian Pilato <christian.pilato@polimi.it>

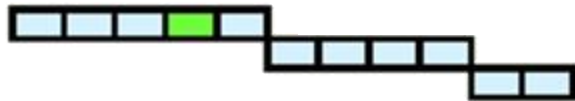
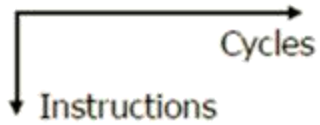
Recall: Parallelism? Which kind?

Instruction-level parallelism (ILP) ←
Data-level parallelism (DLP)
Thread-level parallelism (TLP)
Request-level parallelism (RLP)



Recall: The ILP Architecture Journey

Steps towards exploiting more ILP



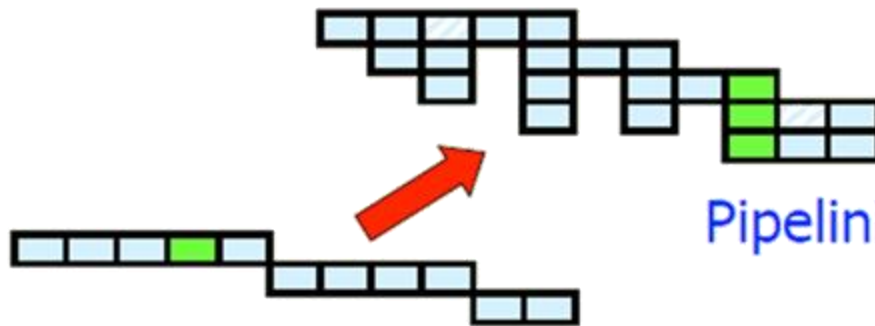
Sequential (non pipelined)

IDEAL CPI > 1

Recall: The ILP Architecture Journey

Steps towards exploiting more ILP

Cycles
Instructions



Pipelining

IDEAL CPI = 1

Sequential (non pipelined)

IDEAL CPI > 1

Recall: Pipeline performance

Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls

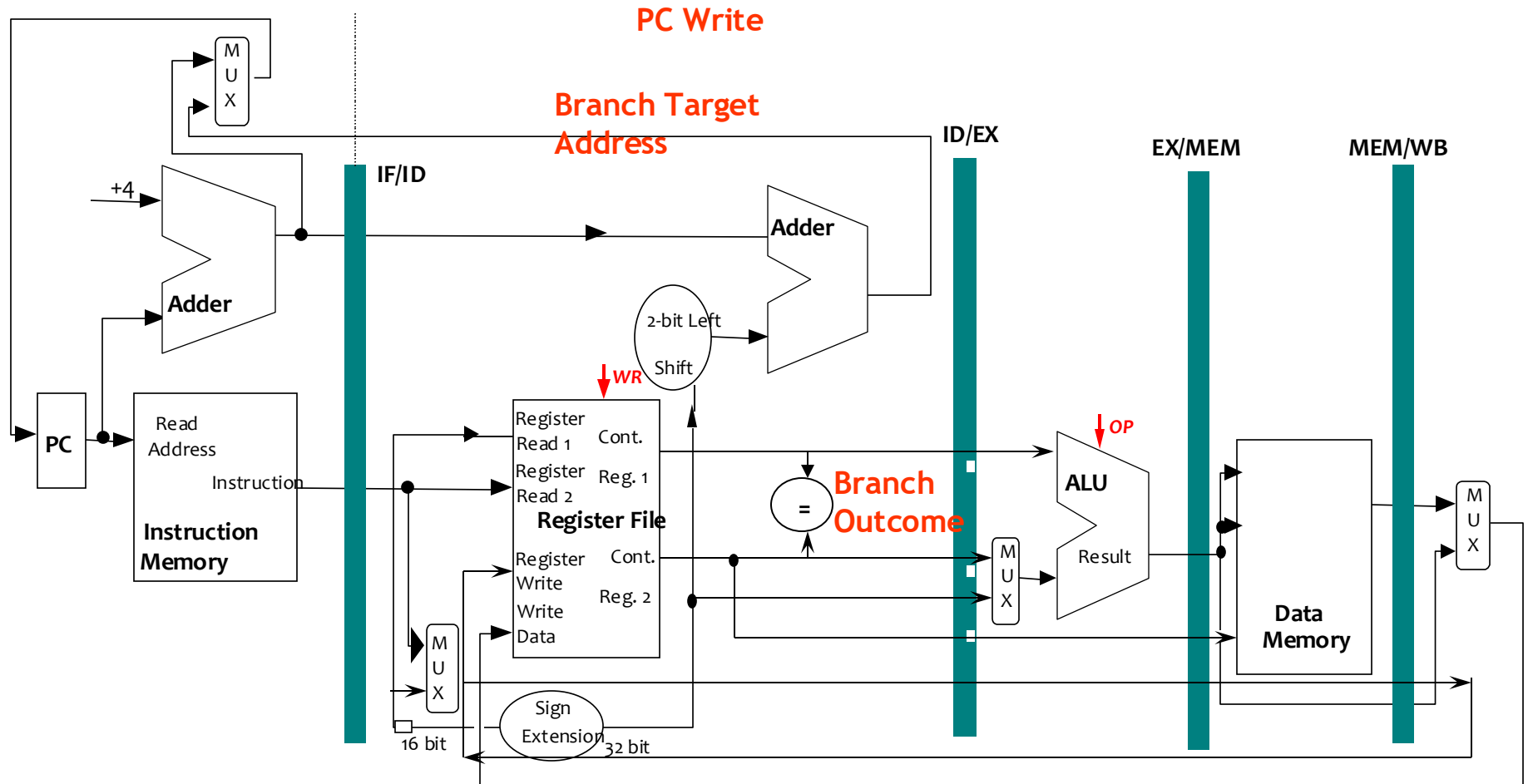
Ideal pipeline CPI: measure of the maximum performance attainable by the implementation

Structural hazards: HW cannot support this combination of instructions

Data hazards: Instruction depends on result of prior instruction still in the pipeline

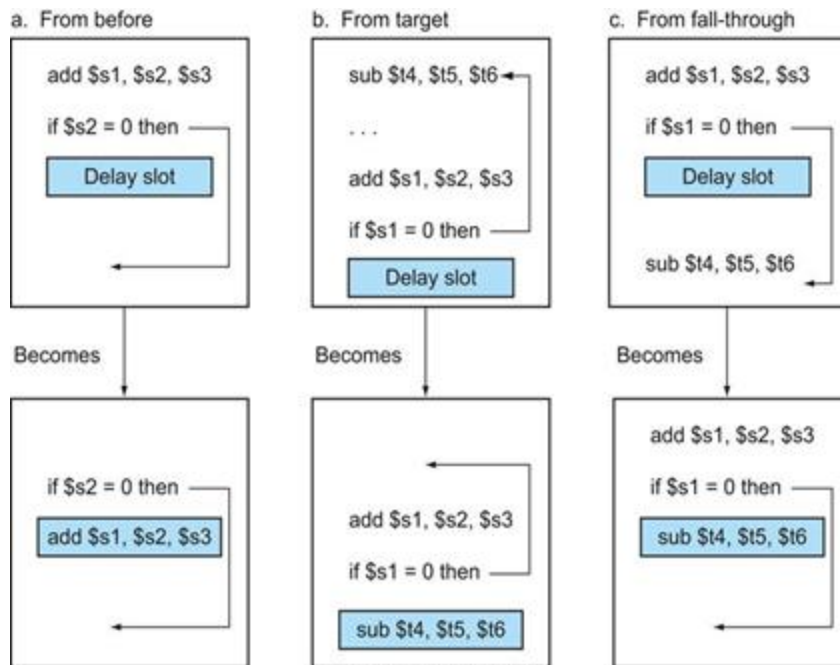
Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches, jumps, exceptions)

Recall: MIPS CPU: Early Evaluation of the PC

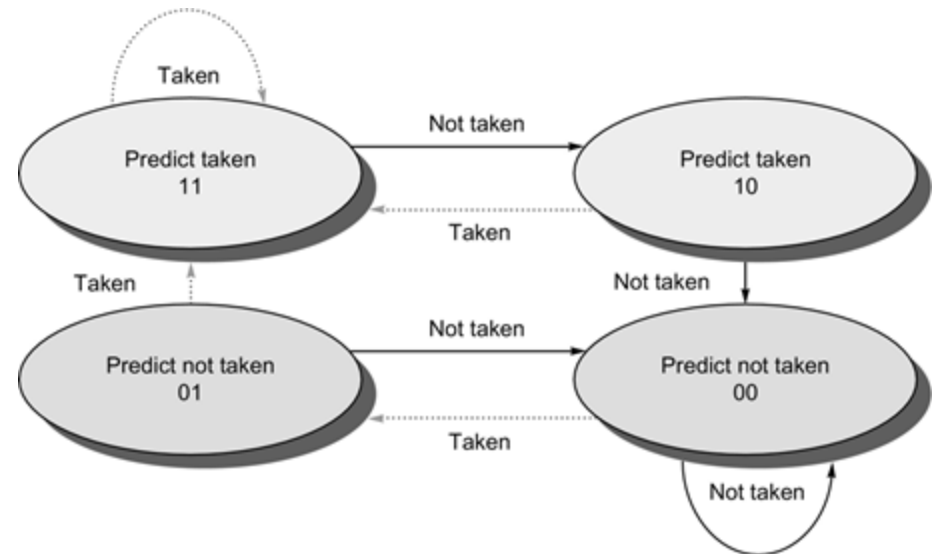


Recall: Branch Prediction

Static Branch Prediction

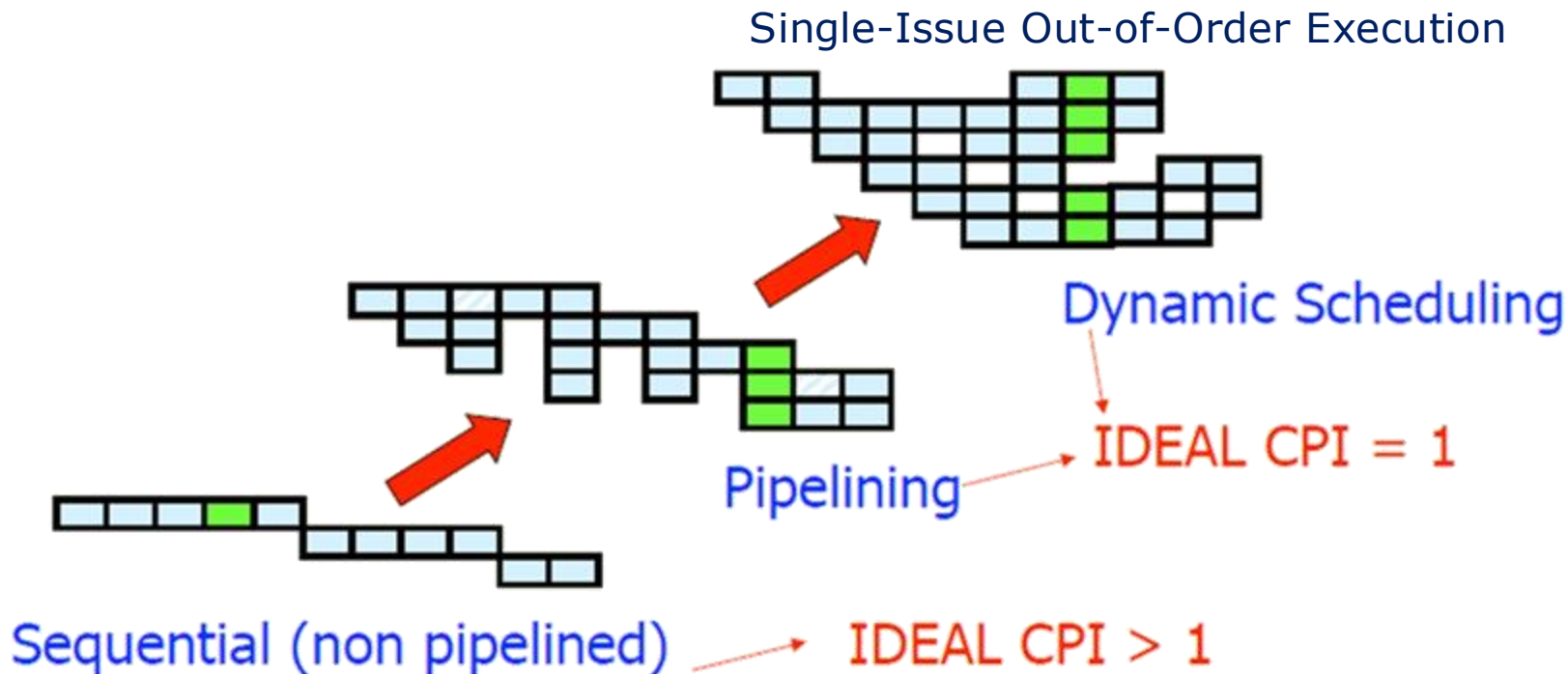
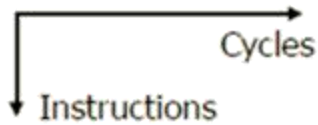


Dynamic Branch Prediction

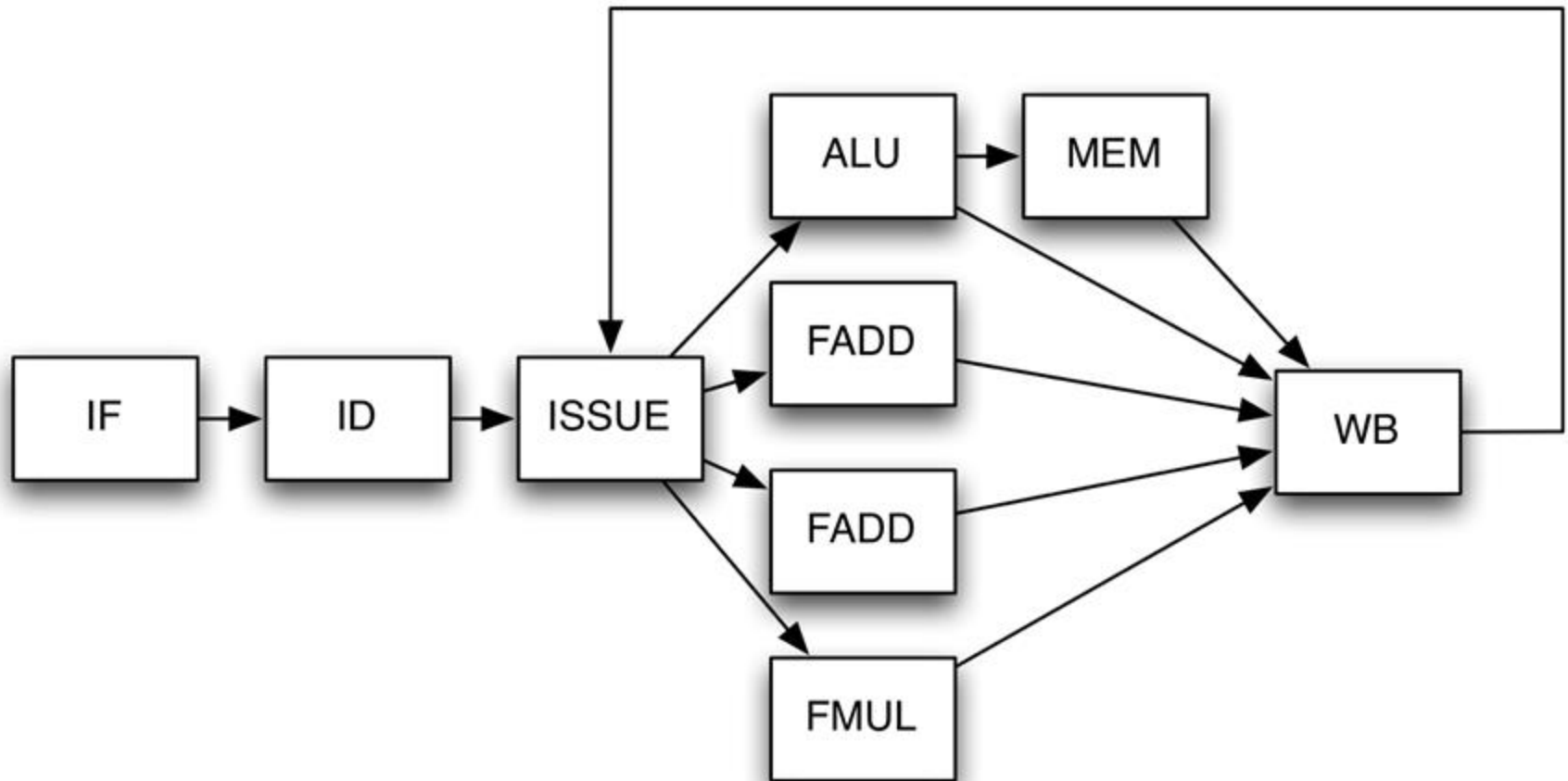


Recall: The ILP Architecture Journey

Steps towards exploiting more ILP



Recall: Example of Complex Pipeline



Recall: Schedule

I	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18
1	F	D	IS	ALU	M	W												
2		F	S(ID)	S(ID)	D	IS	FA	FA	W									
3			S(IF)	S(IF)	F	D	S(IS)	S(IS)	IS	FA	FA	W						
4						F	S(ID)	S(ID)	S(ID)	S(ID)	D	IS	FA	FA	W			
5							S(IF)	S(IF)	S(IF)	S(IF)	F	D	S(IS)	S(IS)	IS	ALU	M	W

Because of the **RAW** we can enter the ISSUE stage at cc15

```
lw    $f1, BASEA($r1)
add.d $f1, $f1, $f5
add.d $f3, $f2, $f1
add.d $f1, $f2, $f3
sw.d  $f1, BASEA($r1)
```

RAW f1, **WAW** f1, **WAR** f1
RAW f1
RAW f3, **WAW** f1, **WAR** f1
RAW f1

Recall: Schedule

I	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18
1	F	D	IS	ALU	M	W												
2		F	S(ID)	S(ID)	D	IS	FA	FA	W									
3			S(IF)	S(IF)	F	S(IS)	S(IS)	IS	FA	FA	W							
4						F	S(ID)	S(ID)	S(ID)	D	IS	FA	FA	W				
5							S(IF)	S(IF)	S(IF)	S(IF)	F	D	S(IS)	S(IS)	IS	ALU	M	W

Because of the **RAW** we can enter the **ISSUE** stage at cc15

```
lw    $f1, BASEA($r1)
add.d $f1, $f1, $f5
add.d $f3, $f2, $f1
add.d $f1, $f2, $f3
sw.d  $f1, BASEA($r1)
```

RAW f1, **WAW** f1, **WAR** f1
RAW f1
RAW f3, **WAW** f1, **WAR** f1
RAW f1

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

Outline

- Introduction to ILP (a short reminder)
- VLIW architecture
- Static Scheduling
 - Basic Blocks
 - Trace scheduling

What's ILP?

What's ILP

Architectural technique that allows the overlap of individual machine operations (add, mul, load, store ...)

Multiple operations will execute in parallel (simultaneously)

Goal: Speed Up the execution

ILP vs Parallel Processing

ILP

- Overlap individual machine operations (add, mul, load...) so that they execute in parallel
- Transparent to the user
- Goal: speed up execution

Parallel Processing

- Having separate processors getting separate chunks of the program (processors programmed to do so)
- Nontransparent to the user
- Goal: speed up and quality up

Beyond CPI = 1

Initial goal to achieve CPI = 1
Can we improve beyond this?

Two approaches

Superscalar and VLIW



based on dynamic scheduling



based on static scheduling

Beyond CPI = 1

Initial goal to achieve CPI = 1
Can we improve beyond this?

Two approaches

Superscalar and **VLIW**

Beyond CPI = 1

--> our goal.

How to have more than one instruction per clock cycles? We should fetch multiple instruction per clock cycle.

With VLIW we actually fetch a single long instruction word that contains multiple instructions

- (Very) Long Instruction Words (V)LIW:
 - fixed number of instructions (4-16)
 - scheduled by the compiler; put ops into wide templates -> the compiler decides how to bundle instructions
 - Currently found more success in DSP, Multimedia applications
 - Joint HP/Intel agreement in 1999/2000
 - Intel Architecture-64 (Merced/A-64) 64-bit address
 - Style: “Explicitly Parallel Instruction Computer (EPIC)”
- ILP with a CPI > 1, where to start from...

Very Long Instruction Word Architectures

Processor can initiate multiple operations per cycle
Specified completely by the compiler (!like
superscalar)

(since everything is solved by compiler)
Low hardware complexity (no scheduling hardware,
reduced support of variable latency instructions)
No instruction reordering performed by the hardware

Explicit parallelism

Single control flow

The idea of VLIW is providing a free dependency code to the architecture. That is, what is going to be fetched will not contain dependencies and so doesn't have to be checked

VLIW processors

How to go from something that does have dependencies into a free-dependency code?

Operation vs instruction

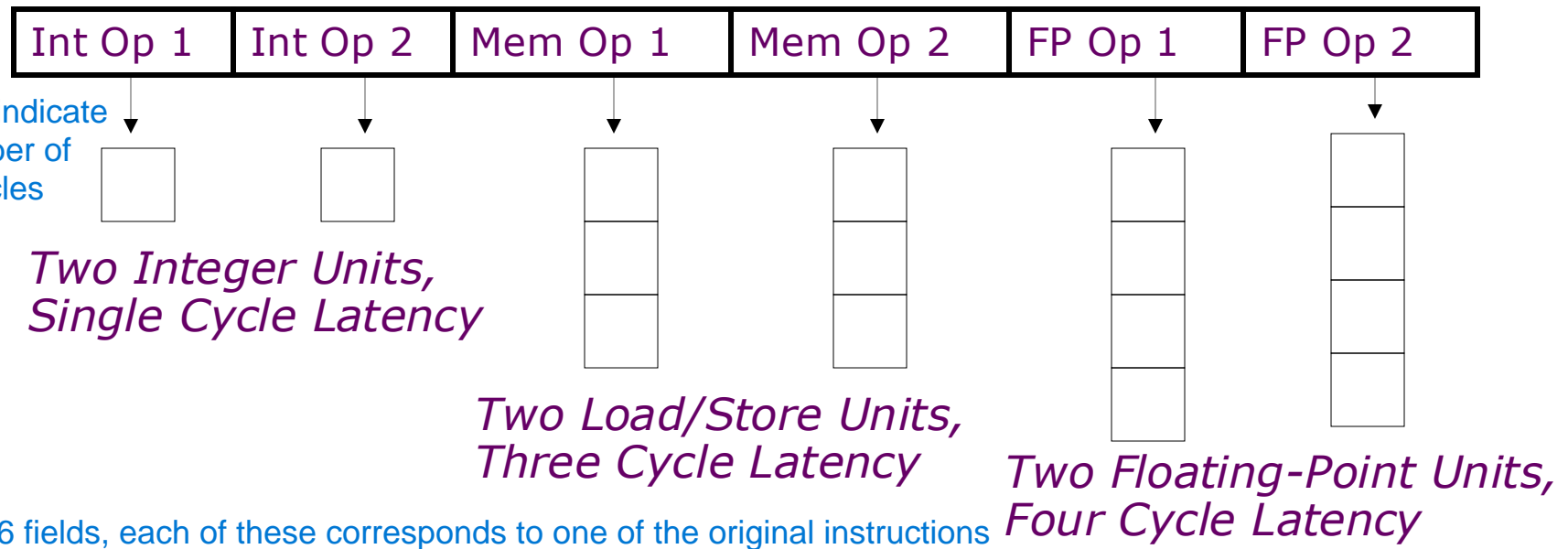
Operation: is a unit of computation (add, load, branch = instruction in sequential ar.)

Instruction: set of operations that are intended to be issued simultaneously

Compiler decides which operation to go to each instruction (scheduling)

All operations that are supposed to begin at the same time are packaged into a single VLIW instruction

VLIW: Very Long Instruction Word



Multiple operations packed into one instruction

Each operation **slot** is for a fixed function

Constant operation latencies are specified

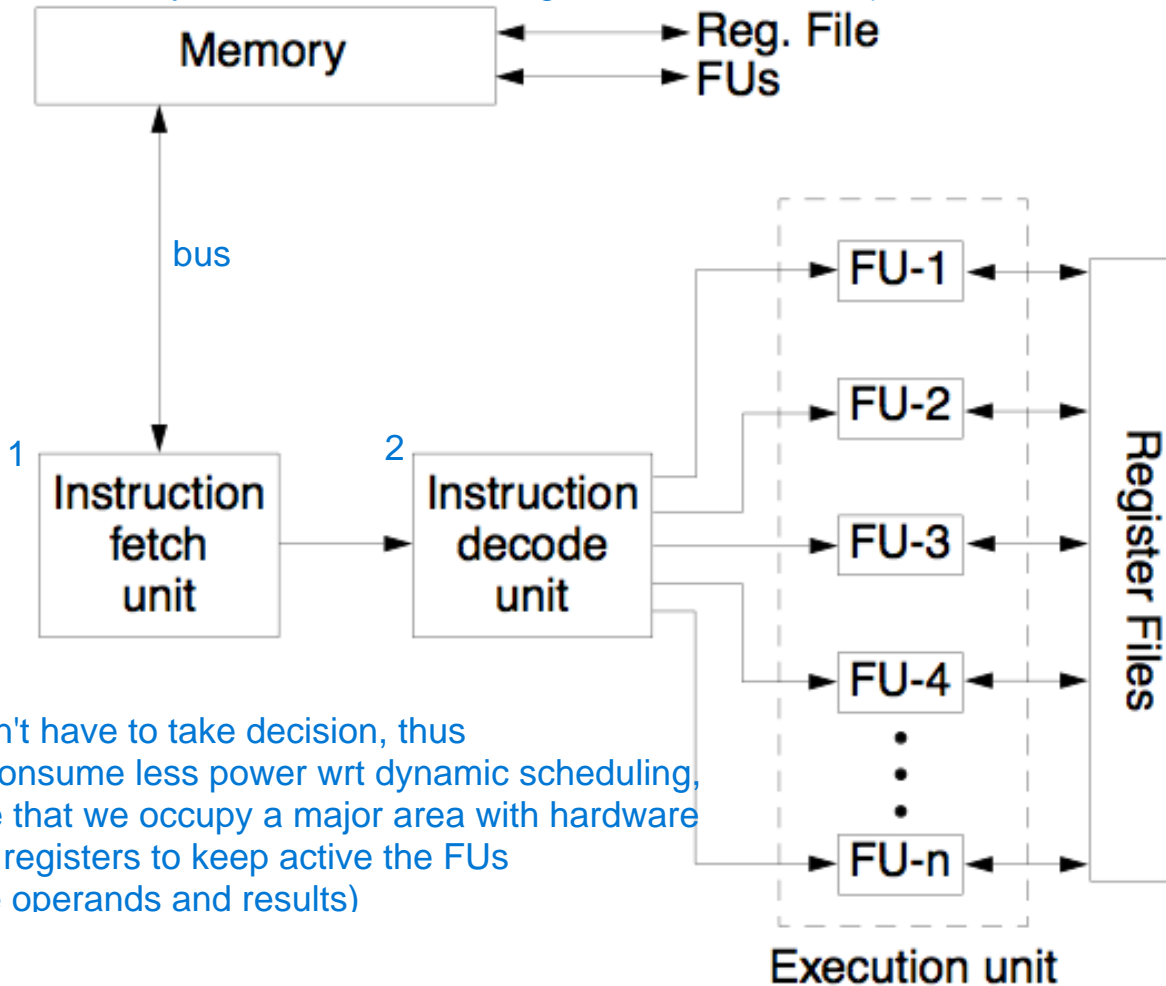
Architecture requires guarantee of:

Parallelism within an instruction => no x-operation RAW check

No data use before data ready => no data interlocks

A VLIW Machine Configuration

We create a new instruction that is going to be fetched in (1), we decode each of the operation inside the big instruction in (2), then we feed all the Functional Units with the operation that are contained in the instruction. That is the reason why the structure of the big instruction is fixed (because it reflect the number of Functional Units)



At runtime we don't have to take decision, thus we are going to consume less power wrt dynamic scheduling, even if it could be that we occupy a major area with hardware (Huge number of registers to keep active the FUs
• Needed to store operands and results)

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 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

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

Instruction Level Parallelism: Today's focus

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

Static Scheduling: the idea

Try to keep pipeline full (in single issue pipelines) or utilize all FUs in each cycle (in VLIW) as much as possible to reach better ILP and therefore higher parallel speedups.

Static Scheduling: general context

Compilers can use sophisticated algorithms for code scheduling to exploit ILP (Instruction Level Parallelism).

single stream of instruction: block of instruction without branches

The amount of parallelism available within a basic block is quite small.

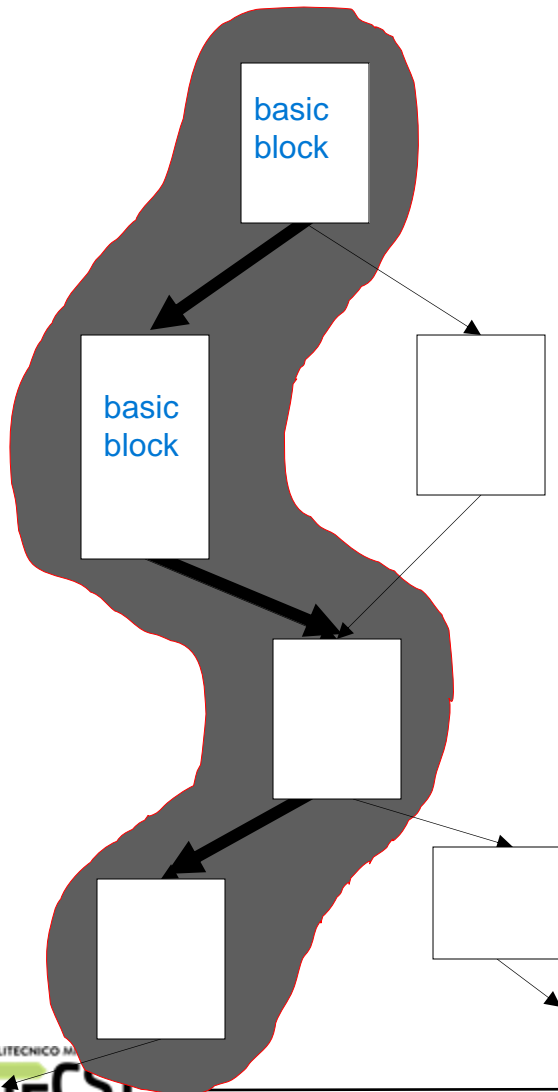
Data dependence can further limit the amount of ILP we can exploit within a basic block to much less than the average basic block size.

To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks (i.e. across branches).

Static Scheduling: general context

Compilers can use sophisticated algorithms for code scheduling to exploit ILP (Instruction Level Parallelism).

Trace Scheduling [Fisher, Ellis]



Ellis JR (1985) Bulldog: a compiler for VLIW architectures. PhD thesis, Yale University
<https://www.dropbox.com/s/kylqrvqcqi137qfe/tr364.pdf?dl=0>

Fisher JA (1993) Global code generation for instruction-level parallelism: trace scheduling-2. Technical Report HPL-93-43. Hewlett-Packard Laboratories
<https://www.dropbox.com/s/0c6go3udnppq3ov/10.1.1.474.6658.pdf?dl=0>

Fisher JA (1981) Trace scheduling: a technique for global microcode compaction, IEEE Trans Comput, July 1981, 30(7):478–490
<https://www.dropbox.com/s/m5j0lmy47qyghe4/TraceScheduling.pdf?dl=0>

We are going to see more, not only about Trace Scheduling, next in this class

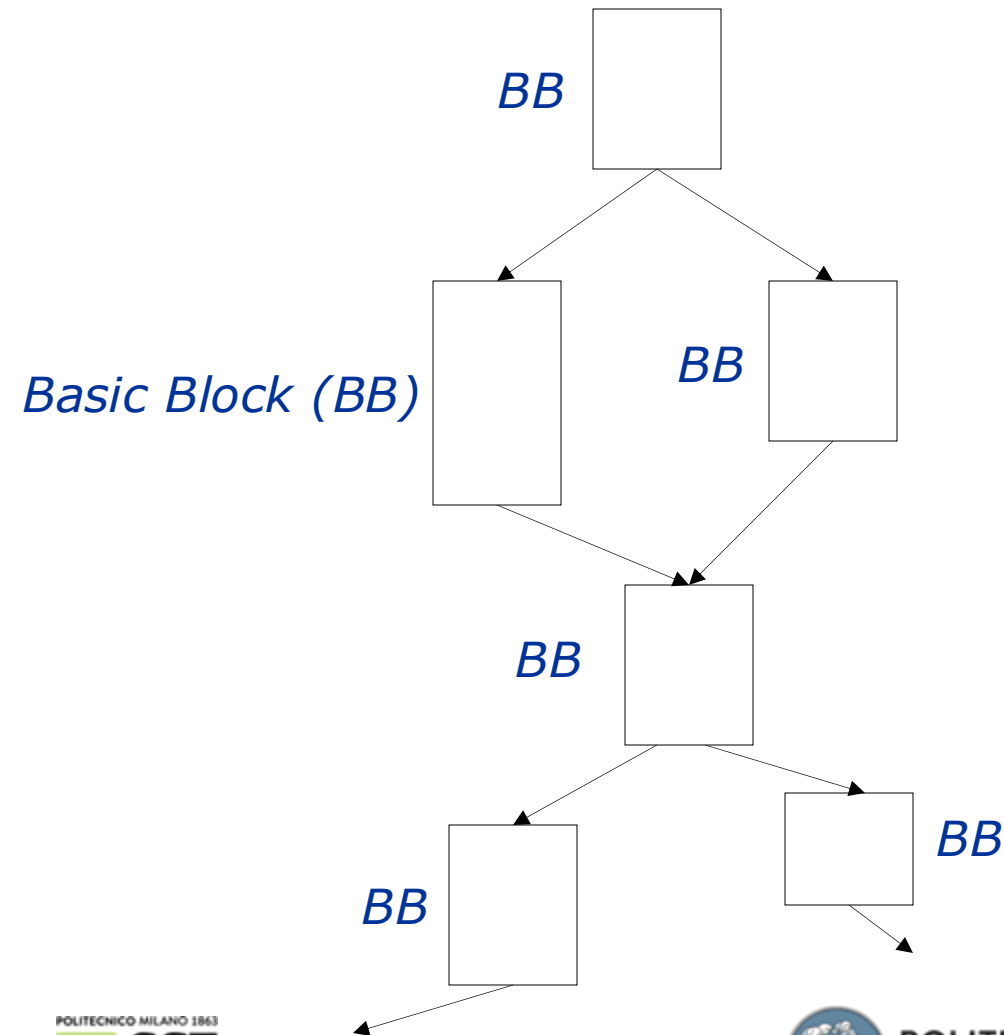
Static Scheduling: general context

Compilers can use sophisticated algorithms for code scheduling to exploit ILP (Instruction Level Parallelism).

The amount of parallelism available within a basic block is quite small. Data dependence can further limit the amount of ILP we can exploit within a basic block to much less than the average basic block size. To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks (i.e. across branches).

Basic Block Definition

Basic block: a sequence of straight non-branch instructions



Detection and resolution of dependences

Static Scheduling

We are going to schedule all the instruction because statically we can't know how the code will behave at runtime

Static detection and resolution of dependences (\Rightarrow **static scheduling**): accomplished by the compiler \Rightarrow dependences are avoided by code reordering.
Output of the compiler: reordered into dependency-free code.

Typical example: VLIW (Very Long Instruction Word) processors expect **dependency-free code**.

VLIW: Pros and Cons

Pros Energy efficiency!

- **Simple HW**
 - Easy to extend the #FUs
- **Good compilers** can effectively detect parallelism

Cons

- **Huge number of registers** to keep active the FUs
 - Needed to store operands and results
- **Large data transport capacity** between
 - FUs and register files
 - Register files and Memory
- **High bandwidth** between **i-cache** and **fetch unit** (instruction we have to fetch)
- **Large code size** (instruction cache)
- **Binary compatibility**

the bigger is the instruction the higher bandwidth we need to move it

we are going to fetch each clock cycle a very long instruction word, that means that even if we are not going to be able to fill the whole long instruction, we will fetch an instruction which contains less instruction than possible.

The more operation the bulde can contian, the larger is the code going to be.

The large the code. the less parallelism I can extract from code. the less efficient I can be.

Knowing branch probabilities

- Profiling requires a significant extra step in build process

Scheduling for statically unpredictable branches

- Optimal schedule varies with branch path

Static Scheduling: methods

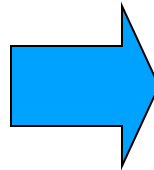
- Simple code motion
- Loop unrolling & loop peeling
- Software pipeline
- Global code scheduling (across basic block)
 - Trace scheduling
 - Superblock scheduling
 - Hyperblock scheduling
 - Speculative Trace scheduling

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

Scheduling: Code Motion

```
sub $2, $1, $3  
and $12, $2, $5  
or $13, $6, $2  
add $14, $2, $2  
sw $15, 100($2)  
add $4, $10, $11  
and $7, $8, $9  
lw $16, 100($18)  
lw $17, 200($19)
```



```
sub $2, $1, $3  
add $4, $10, $11  
and $7, $8, $9  
lw $16, 100($18)  
lw $17, 200($19)  
and $12, $2, $5  
or $13, $6, $2  
add $14, $2, $2  
sw $15, 100($2)
```

Loop Execution

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

Loop Execution

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

↓ *Compile*

```
loop: ld f1, 0(r1)  
      add r1, 8  
      fadd f2, f0, f1  
      sd f2, 0(r2)  
      add r2, 8  
      bne r1, r3, loop
```


Loop Execution

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

↓ *Compile*

```
loop: ld f1, 0(r1)  
      add r1, 8  
      fadd f2, f0, f1  
      sd f2, 0(r2)  
      add r2, 8  
      bne r1, r3, loop
```

ld, sd : 3cc
add, bne : 1cc
fadd: 4cc

Loop Execution

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

Compile

```
loop: ld f1, 0(r1)
      add r1, 8
      fadd f2, f0, f1
      sd f2, 0(r2)
      add r2, 8
      bne r1, r3, loop
```

loop:

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc

N.B. This is for floating point adds

ld, sd : 3cc
add, bne : 1cc
fadd: 4cc

This is for
FP
multiplication
n

Loop Execution

ld, sd : 3cc
add, bne : 1cc
fadd: 4cc

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

Compile

```
loop: ld f1, 0(r1)  
      add r1, 8  
      fadd f2, f0, f1  
      sd f2, 0(r2)  
      add r2, 8  
      bne r1, r3, loop
```

We will create the code that we are going to fetch

loop:

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld			

Loop Execution

ld, sd : 3cc
add, bne : 1cc
fadd: 4cc

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

Compile

```
loop: ld f1, 0(r1) } WAR  
      add r1, 8  
      fadd f2, f0, f1  
      sd f2, 0(r2)  
      add r2, 8  
      bne r1, r3, loop
```

loop:

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
add r1		ld		can we put the fadd here?	
				structural hazards? no it's free dependencies? on f0? no problem on f1 RAW with load	
				Thus, we need to wait the ld to finish	

We don't care about WAR now:

- fetch is done simultaneously
- decode too
- then each instruction is executed simultaneously, that means that both the ld and the add have their own copy of r1 (heres why we have many registers: each instruction has its own): thus the WAR it is not a problem

Loop Execution

ld, sd : 3cc
add, bne : 1cc
fadd: 4cc

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

Compile

```
loop: ld f1, 0(r1)  
      add r1, 8  
      fadd f2, f0, f1  
      sd f2, 0(r2)  
      add r2, 8  
      bne r1, r3, loop
```

loop:

space for 24
instruction but
only 3 instructions

Schedule

	Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
add r1			ld		1cc	
					2cc	
					3cc	

Loop Execution

ld, sd : 3cc
add, bne : 1cc
fadd: 4cc

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

Compile

```
loop: ld f1, 0(r1)  
      add r1, 8  
      fadd f2, f0, f1  
      sd f2, 0(r2)  
      add r2, 8  
      bne r1, r3, loop
```

loop:

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
add r1		ld		1cc	
				2cc	
				3cc	
				fadd	

Loop Execution

ld, sd : 3cc
add, bne : 1cc
fadd: 4cc

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

Compile

```
loop: ld f1, 0(r1)  
      add r1, 8  
      fadd f2, f0, f1  
      sd f2, 0(r2)  
      add r2, 8  
      bne r1, r3, loop
```

loop:

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
add r1		ld			
		1cc		fadd	
		2cc			
		3cc			
		4cc			

Loop Execution

ld, sd : 3cc
add, bne : 1cc
fadd: 4cc

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

Compile

```
loop: ld f1, 0(r1)  
      add r1, 8  
      fadd f2, f0, f1  
      sd f2, 0(r2)  
      add r2, 8  
      bne r1, r3, loop
```

loop:

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
add r1		ld			
		1cc		fadd	
		2cc			
		3cc			
		4cc			
		sd			

Loop Execution

ld, sd : 3cc
add, bne : 1cc
fadd: 4cc

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

Compile

```
loop: ld f1, 0(r1)  
      add r1, 8  
      fadd f2, f0, f1  
      sd f2, 0(r2)  
      add r2, 8  
      bne r1, r3, loop
```

loop:

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
add r1		ld			
				fadd	
add r2		sd			

Loop Execution

ld, sd : 3cc
add, bne : 1cc
fadd: 4cc

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

Compile

```
loop: ld f1, 0(r1)  
      add r1, 8  
      fadd f2, f0, f1  
      sd f2, 0(r2)  
      add r2, 8  
      bne r1, r3, loop
```

loop:

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
add r1		ld			
				fadd	
add r2	bne	sd			

Loop Execution

ld, sd : 3cc
add, bne : 1cc
fadd: 4cc

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

Compile

```
loop: ld f1, 0(r1)  
      add r1, 8  
      fadd f2, f0, f1  
      sd f2, 0(r2)  
      add r2, 8  
      bne r1, r3, loop
```

loop:

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
add r1		ld			
				fadd	
add r2	bne	sd			

How many FP ops/cycle?

Loop Execution

ld, sd : 3cc
add, bne : 1cc
fadd: 4cc

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

Compile

```
loop: ld f1, 0(r1)  
      add r1, 8  
      fadd f2, f0, f1  
      sd f2, 0(r2)  
      add r2, 8  
      bne r1, r3, loop
```

loop:

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
add r1		ld			
				fadd	
add r2	bne	sd			

How many FP ops/cycle?

$$1 \text{ fadd} / 8 \text{ cycles} = 0.125$$

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

Loop Unrolling

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

Loop Unrolling

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



Unroll inner loop to perform 4 iterations at once

Loop Unrolling

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



Unroll inner loop to perform 4 iterations at once

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

Need to handle values of N that are not multiples of unrolling factor with final cleanup loop

Scheduling Loop Unrolled Code

Unroll 4 ways

```
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
```

loop:

Schedule

Int1	Int 2	M1	M2	FP+	FPx
1cc	1cc	3cc	3cc	4cc	4cc

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, loop
    
```

loop:

Schedule →

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

loop:

Schedule →

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1		1cc	
				2cc	
				3cc	

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

loop:

Schedule →

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1		1cc	
				2cc	
				3cc	
				fadd f5	

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

loop:

Schedule →

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		1cc		fadd f5	
		2cc			
		3cc			
		4cc			

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

Schedule

loop:

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		1cc		fadd f5	
		2cc			
		3cc			
		4cc			
		sd f5			

Scheduling Loop Unrolled Code

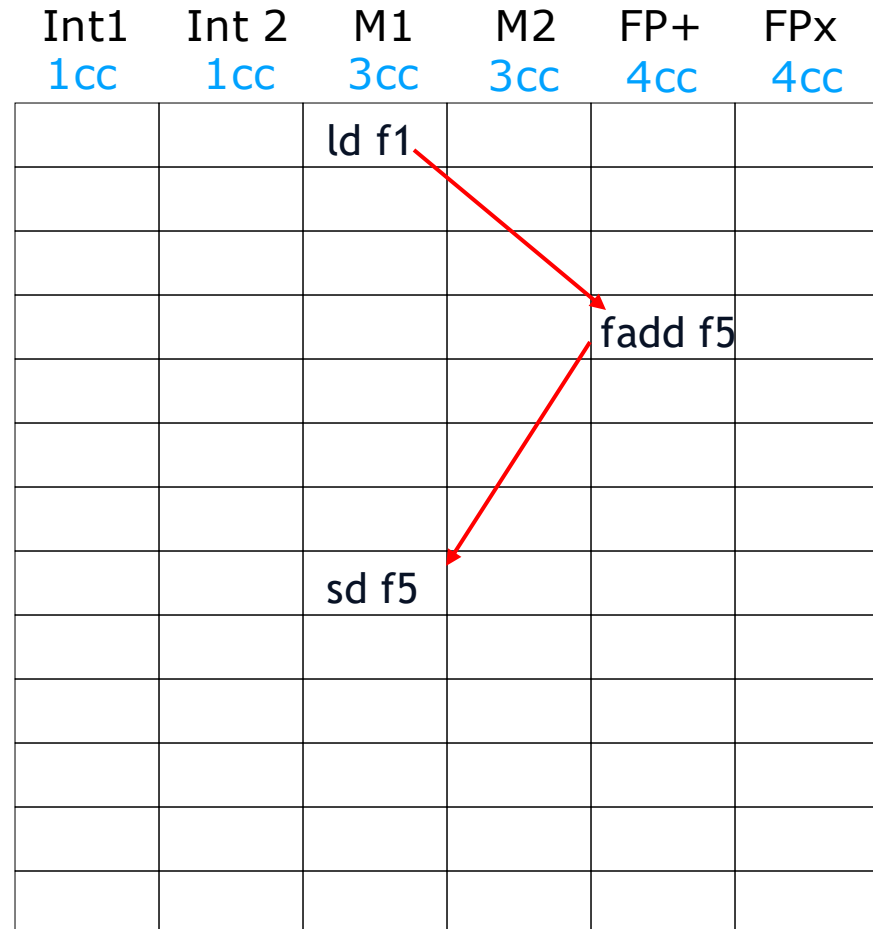
Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

Schedule

loop:



Scheduling Loop Unrolled Code

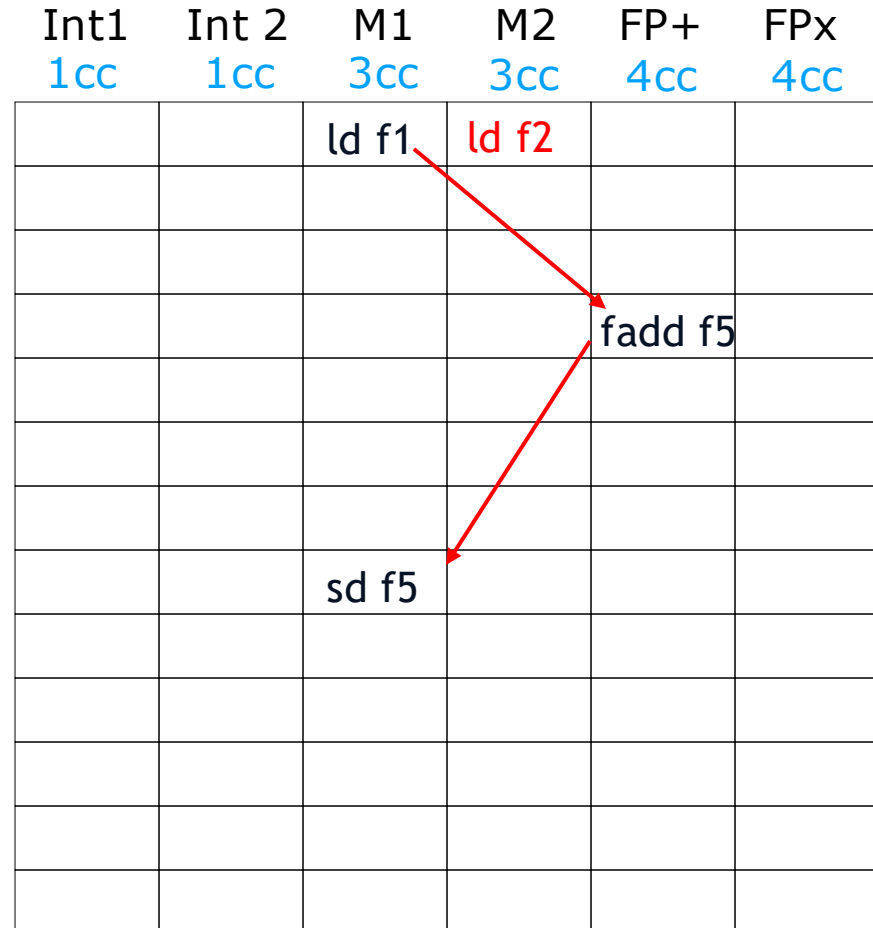
Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

loop:

Schedule



Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, loop
    
```

loop:

Which one?

Schedule →

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1	ld f2		
		ld f2			
				fadd f5	
		sd f5			

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

Actually they are both legit

loop:

Which one?

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1	ld f2		
		ld f2			
				fadd f5	
				fadd f6	
		sd f5			

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

loop:

THIS one!!!

Because we can preserve the scheme.
In any case the other choice would have
not assured better performances

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
				fadd f5	
				fadd f6	
		sd f5			

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

loop:

Schedule →

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
				fadd f5	
				fadd f6	
		sd f5			
		sd f6			

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

loop:

Schedule →

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
				fadd f5	
				fadd f6	
				fadd f7	
		sd f5			
		sd f6			
		sd f7			

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

loop:

Schedule →

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
		ld f4		fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
		sd f5			
		sd f6			
		sd f7			
		sd f8			

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

loop:

ASAP

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4		fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
		sd f5			
		sd f6			
		sd f7			
		sd f8			

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

loop:

Schedule →

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4		fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
		sd f5			
		sd f6			
		sd f7			
add r2		sd f8			

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

loop:

Schedule →

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4		fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
		sd f5			
		sd f6			
		sd f7			
add r2	bne	sd f8			

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

loop:

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4		fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
		sd f5			
		sd f6			
		sd f7			
add r2	bne	sd f8			

How many FLOPS/cycle?

Scheduling Loop Unrolled Code

Unroll 4 ways

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
    
```

loop:

Schedule

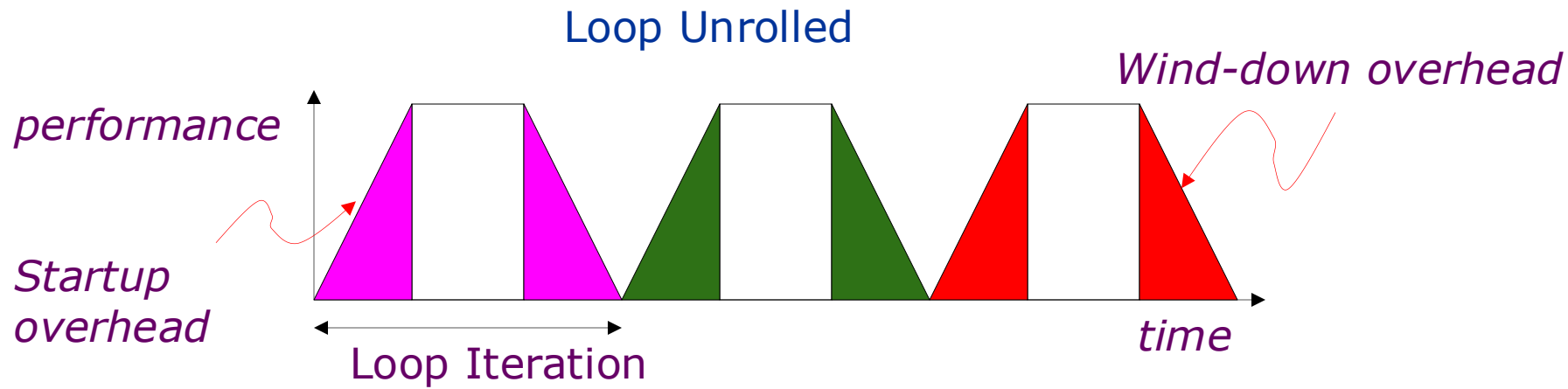
Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4		fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
		sd f5			
		sd f6			
		sd f7			
add r2	bne	sd f8			

How many FLOPS/cycle?

(Floating point
operations)

4 fadds / 11 cycles = 0.36

Loop Unrolling



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

Pipelining meant being able to overlap execution with stages

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop

```

Schedule

MILANO 1863

Software Pipelining

Unroll 4 ways first

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
    
```

loop:

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4			
				fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
			sd f5		
			sd f6		
			sd f7		
	bne		sd f8		

Software Pipelining

Unroll 4 ways first

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
    
```

Schedule

loop:

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4			
				fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
			sd f5		
			sd f6		
			sd f7		
	bne		sd f8		

changes applied by compiler



Software Pipelining

Unroll 4 ways first

```

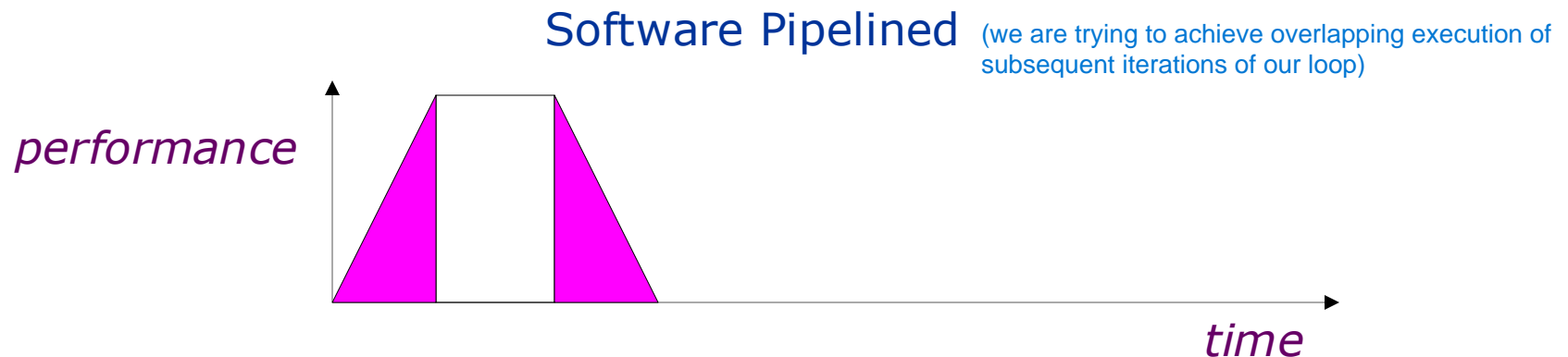
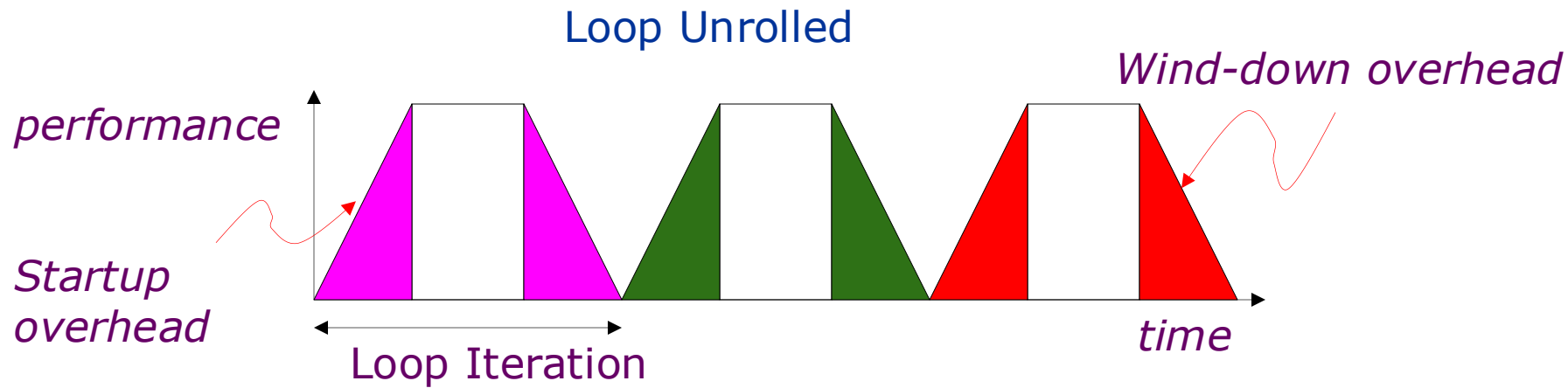
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
    
```

loop:

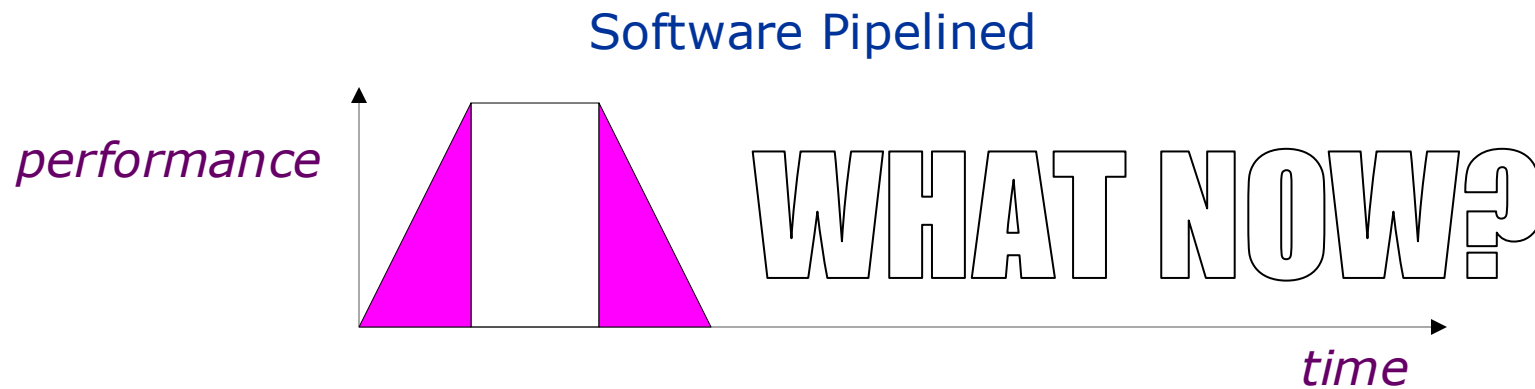
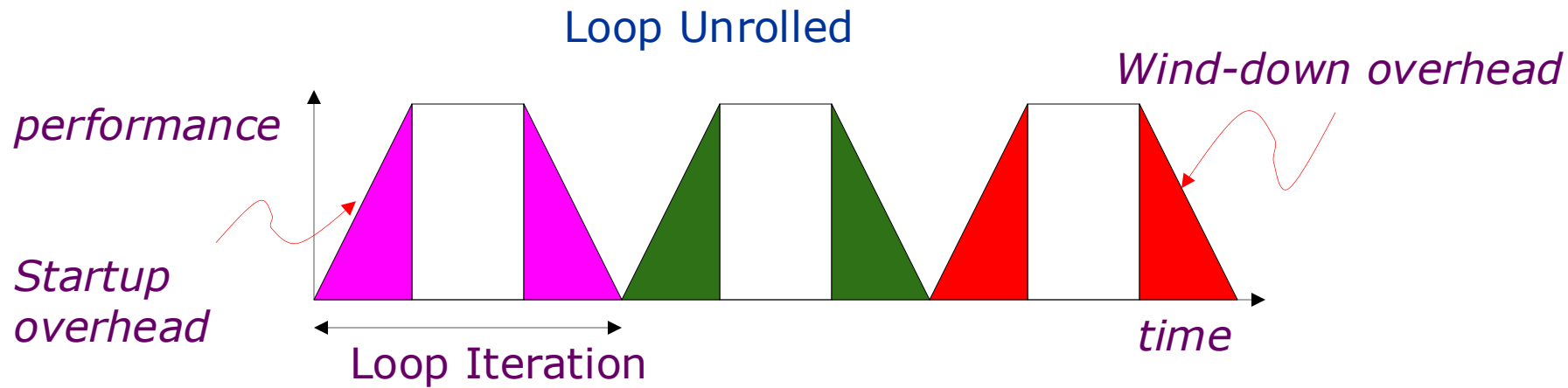
Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4			
				fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
			sd f5		
			sd f6		
	add r2		sd f7		
	bne		sd f8		

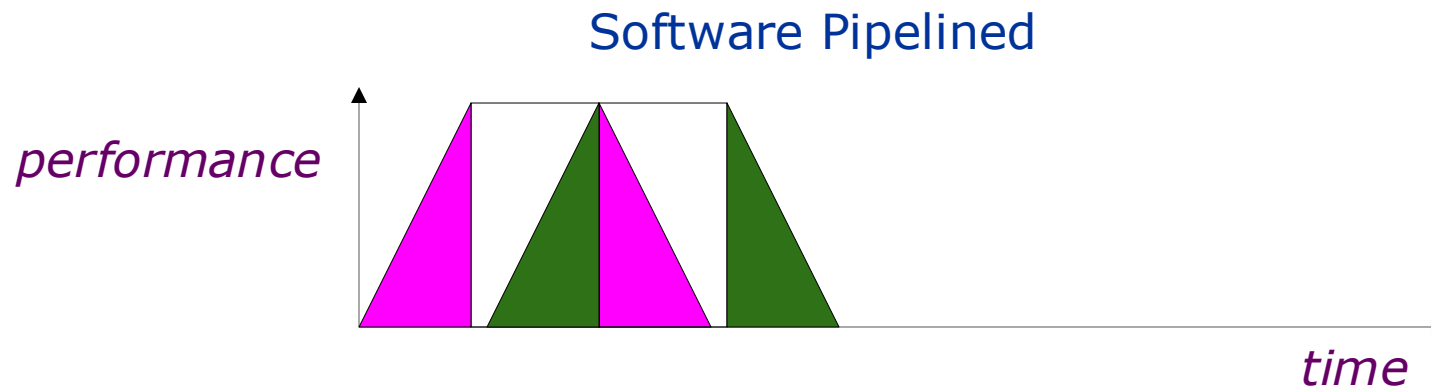
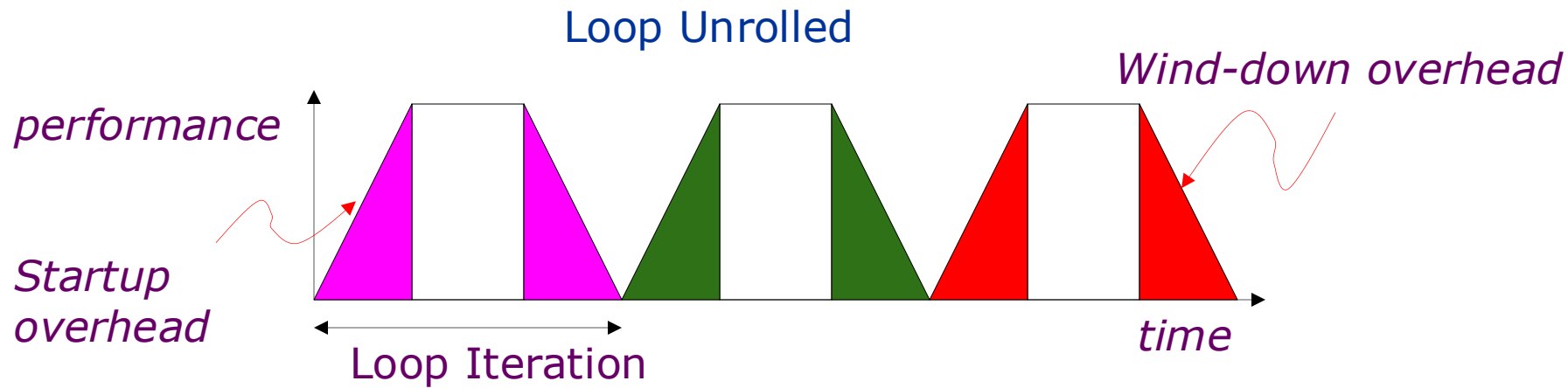
Loop Unrolling vs. Software Pipelining



Loop Unrolling vs. Software Pipelining



Loop Unrolling vs. Software Pipelining



Software Pipelining

Unroll 4 ways first

```

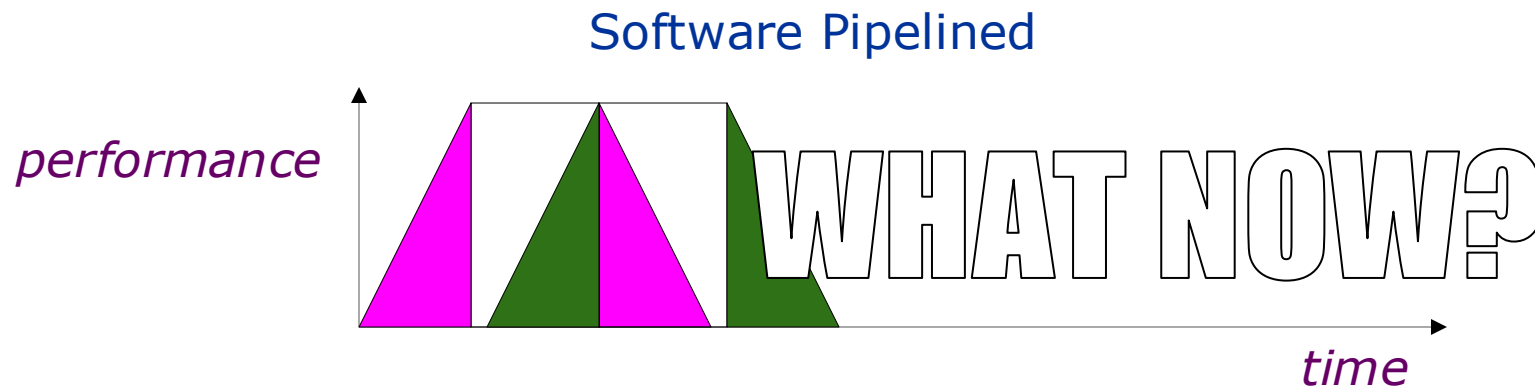
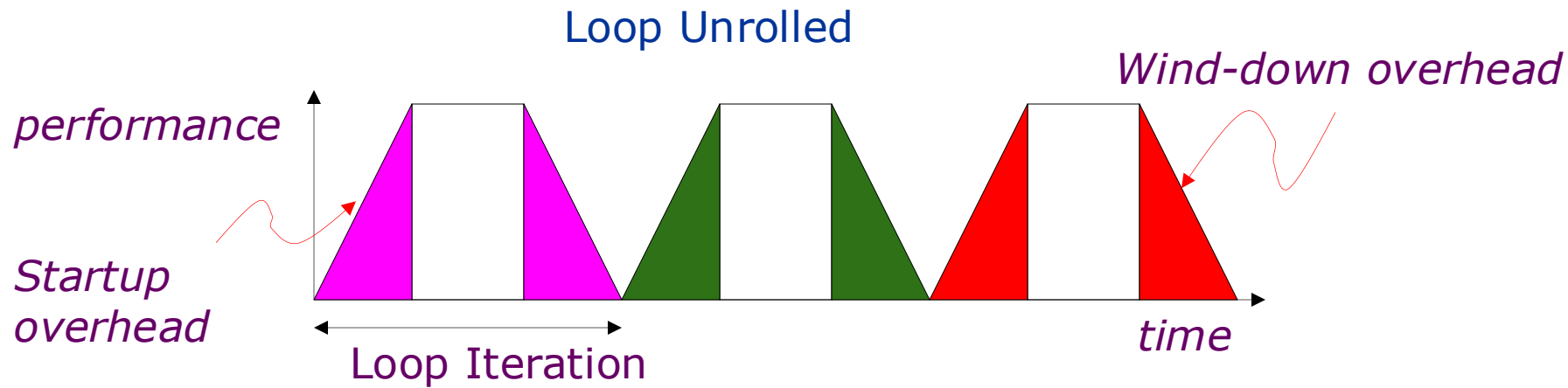
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
    
```

loop:

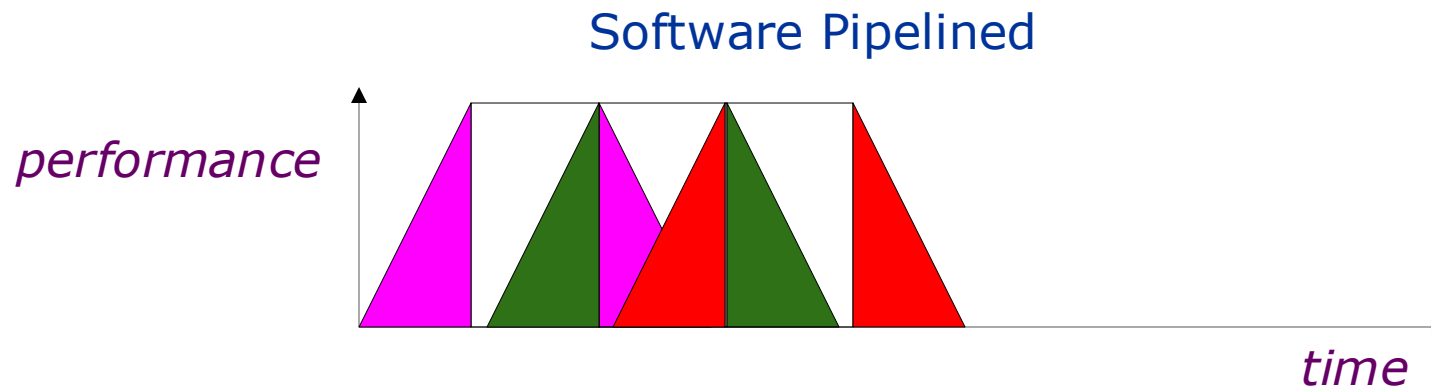
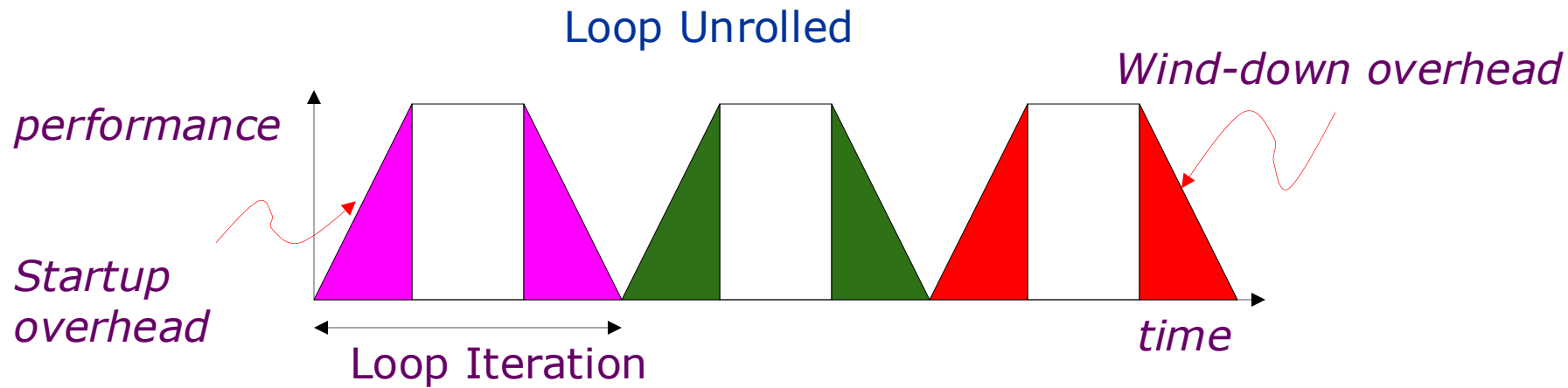
Schedule →

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4			
		ld f1		fadd f5	
		ld f2		fadd f6	
		ld f3		fadd f7	
add r1		ld f4		fadd f8	
			sd f5	fadd f5	
			sd f6	fadd f6	
	add r2		sd f7	fadd f7	
	bne		sd f8	fadd f8	
			sd f5		
			sd f6		
	add r2		sd f7		
	bne		sd f8		

Loop Unrolling vs. Software Pipelining



Loop Unrolling vs. Software Pipelining



Software Pipelining

Unroll 4 ways first

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
    
```

loop:

Schedule →

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4			
		ld f1		fadd f5	
		ld f2		fadd f6	
		ld f3		fadd f7	
add r1		ld f4		fadd f8	
		ld f1	sd f5	fadd f5	
		ld f2	sd f6	fadd f6	
	add r2	ld f3	sd f7	fadd f7	
add r1	bne	ld f4	sd f8	fadd f8	
			sd f5	fadd f5	
			sd f6	fadd f6	
	add r2		sd f7	fadd f7	
	bne		sd f8	fadd f8	

sd f5

Software Pipelining

Unroll 4 ways first

```

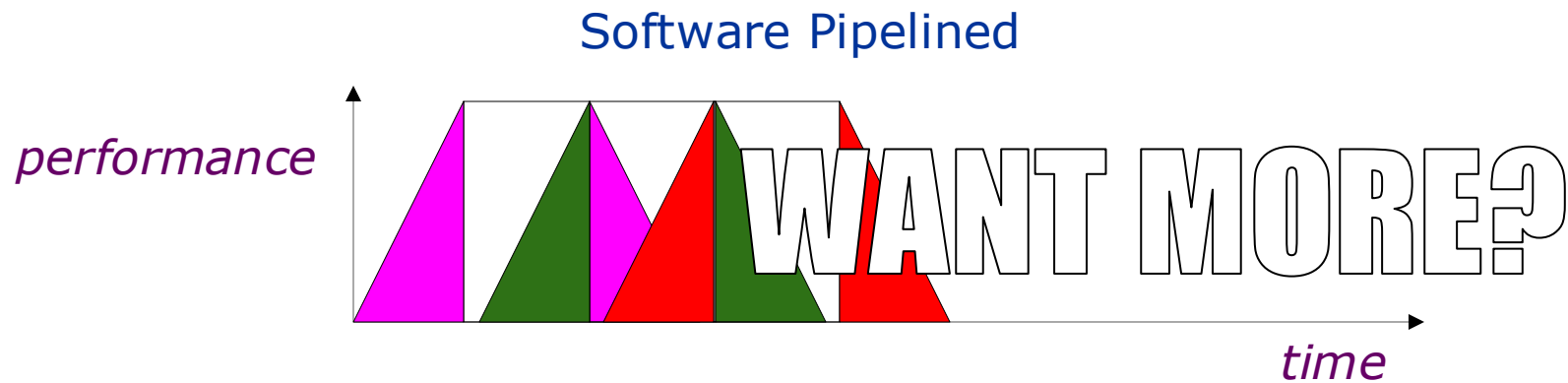
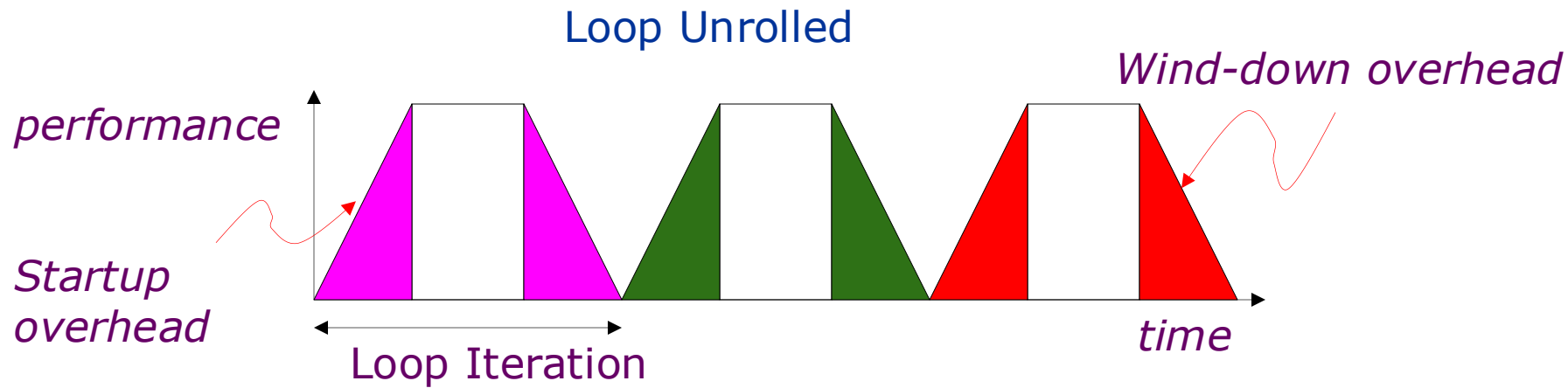
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
    
```

loop:

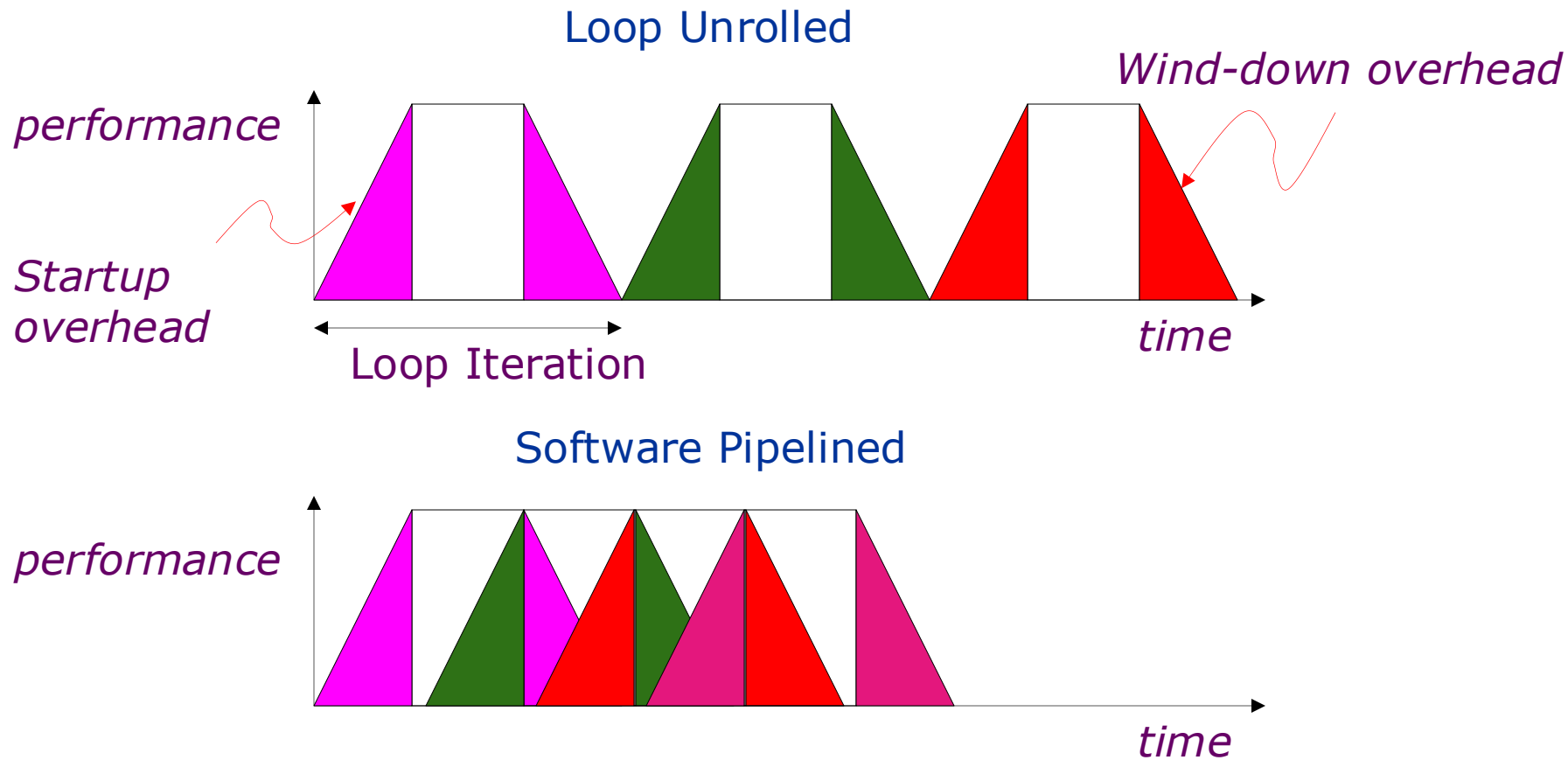
Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4			
		ld f1		fadd f5	
		ld f2		fadd f6	
		ld f3		fadd f7	
add r1		ld f4		fadd f8	
		ld f1	sd f5	fadd f5	
		ld f2	sd f6	fadd f6	
	add r2	ld f3	sd f7	fadd f7	
add r1	bne	ld f4	sd f8	fadd f8	
			sd f5	fadd f5	
			sd f6	fadd f6	
	add r2		sd f7	fadd f7	
	bne		sd f8	fadd f8	
			sd f5		

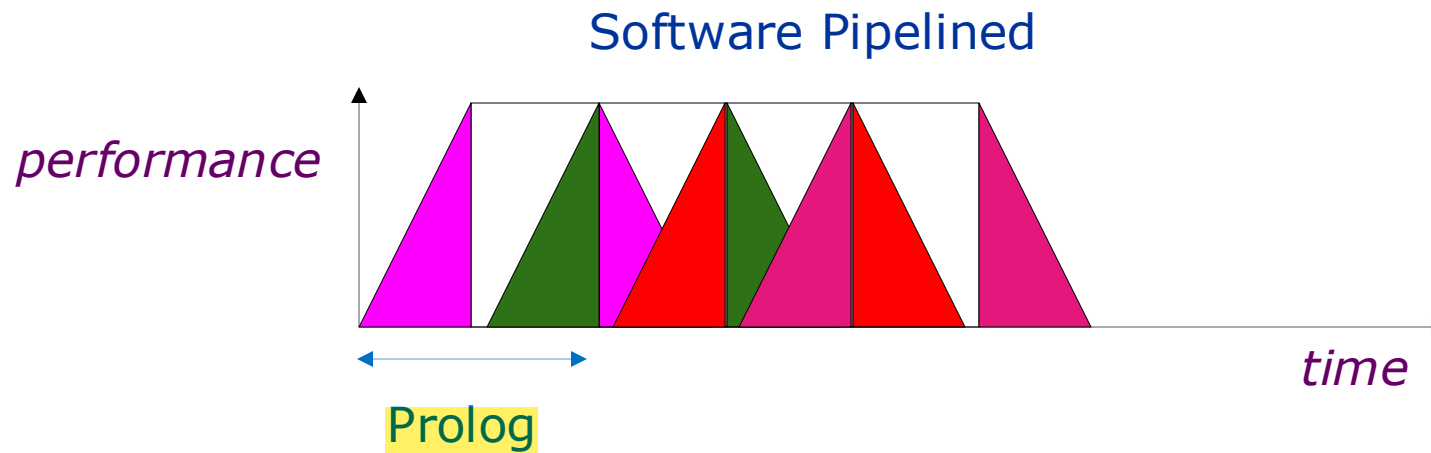
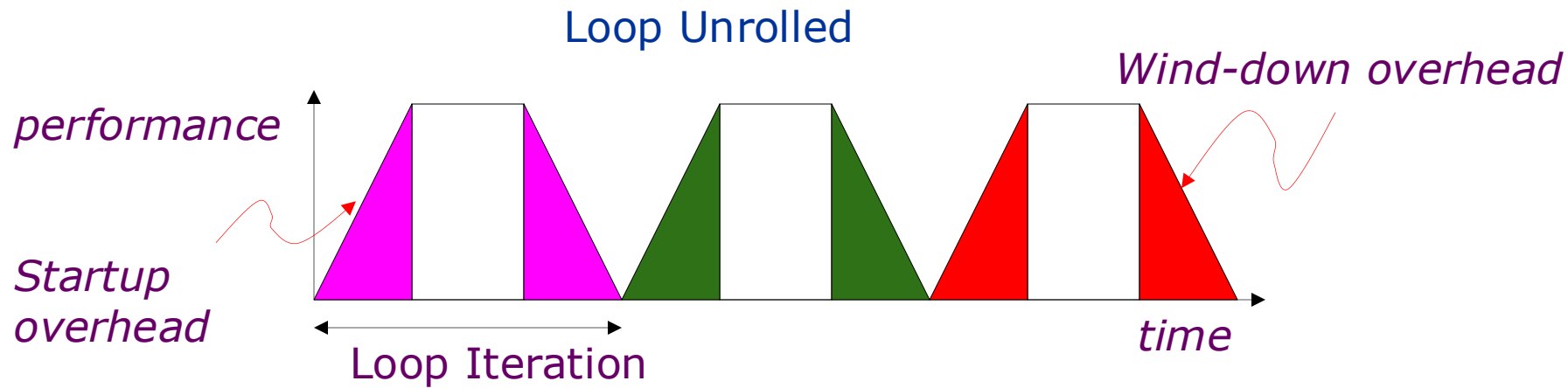
Loop Unrolling vs. Software Pipelining



Loop Unrolling vs. Software Pipelining



Loop Unrolling vs. Software Pipelining



Software Pipelining

Unroll 4 ways first

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
    
```

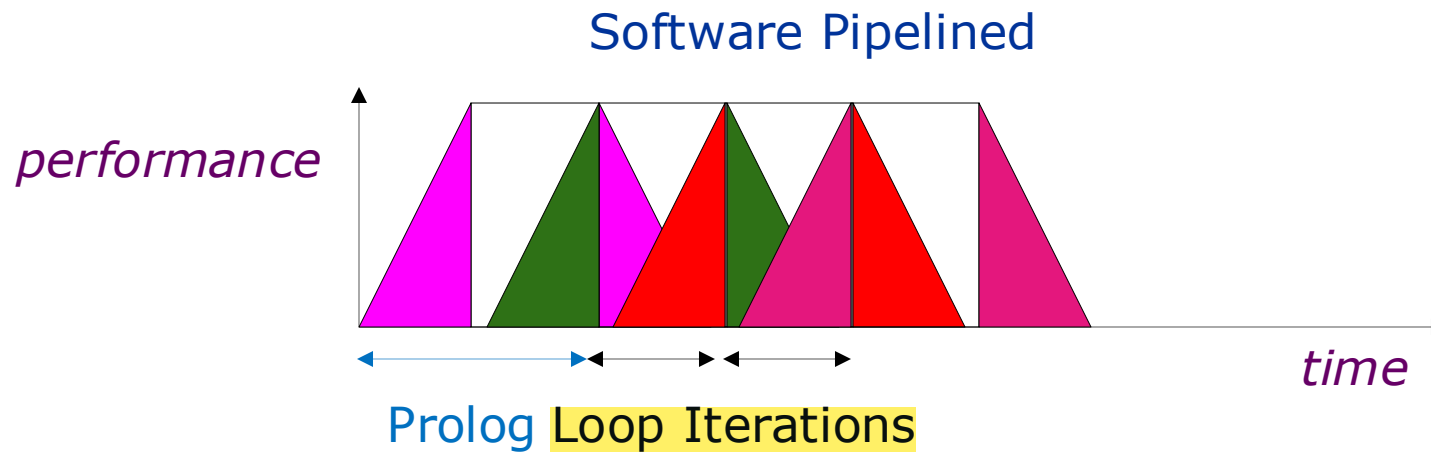
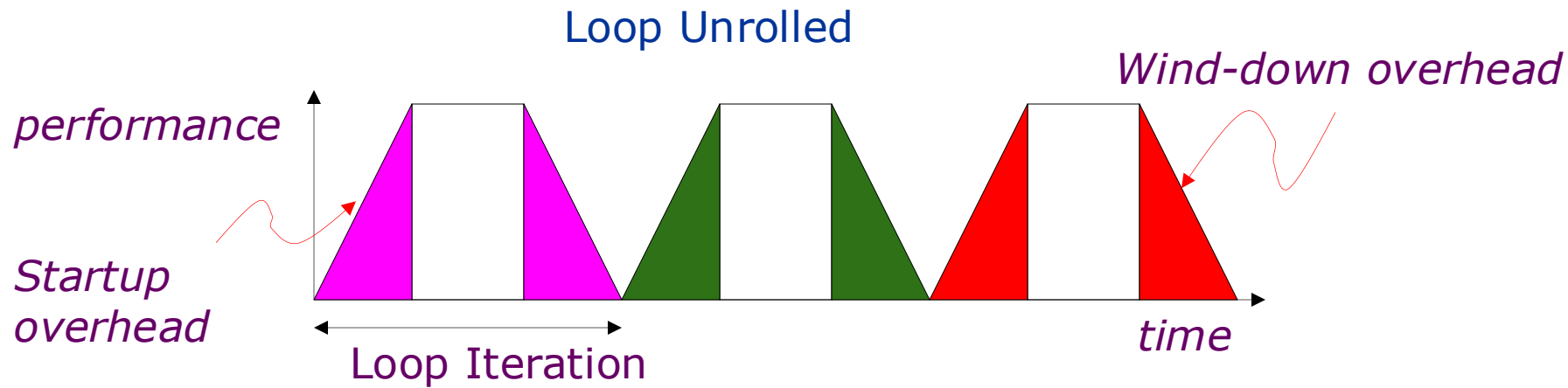
Prolog

Schedule

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4			
		ld f1		fadd f5	
		ld f2		fadd f6	
		ld f3		fadd f7	
add r1		ld f4		fadd f8	
		ld f1	sd f5	fadd f5	
		ld f2	sd f6	fadd f6	
	add r2	ld f3	sd f7	fadd f7	
add r1	bne	ld f4	sd f8	fadd f8	
			sd f5	fadd f5	
			sd f6	fadd f6	
	add r2		sd f7	fadd f7	
	bne		sd f8	fadd f8	

sd f5

Loop Unrolling vs. Software Pipelining



Software Pipelining

Unroll 4 ways first

```

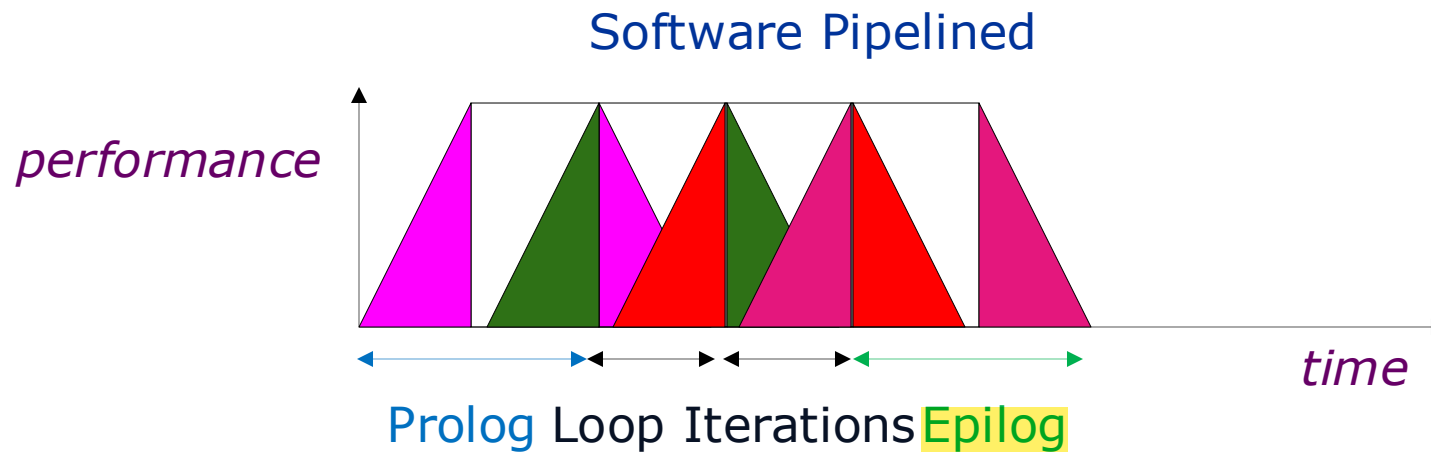
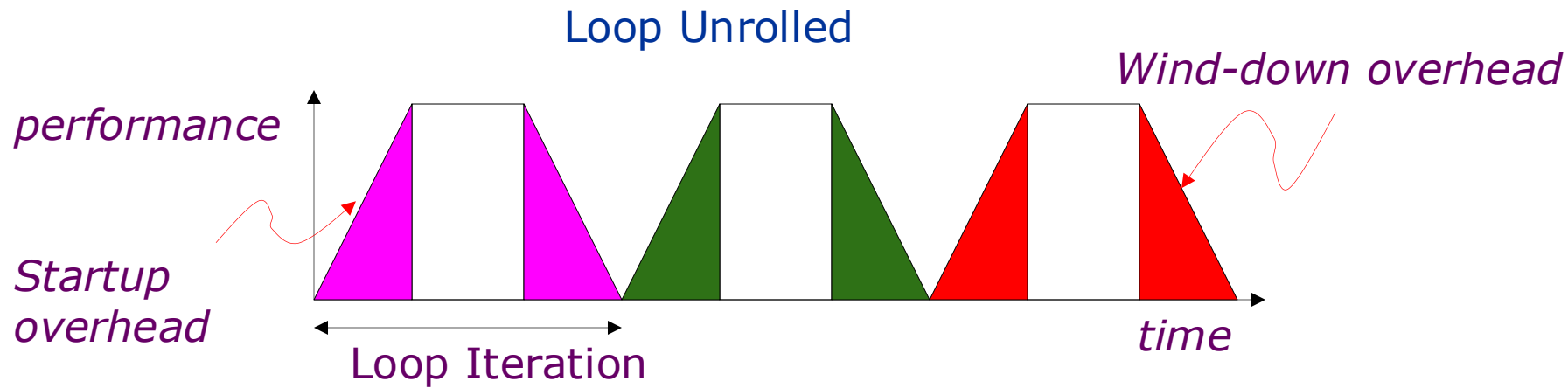
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
    
```

Prolog
Schedule

loop:
iterate

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4			
		ld f1		fadd f5	
		ld f2		fadd f6	
		ld f3		fadd f7	
add r1		ld f4		fadd f8	
		ld f1	sd f5	fadd f5	
		ld f2	sd f6	fadd f6	
	add r2	ld f3	sd f7	fadd f7	
add r1	bne	ld f4	sd f8	fadd f8	
			sd f5	fadd f5	
			sd f6	fadd f6	
	add r2		sd f7	fadd f7	
	bne		sd f8	fadd f8	
			sd f5		

Loop Unrolling vs. Software Pipelining



Software Pipelining

Unroll 4 ways first

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
    
```

Prolog
Schedule

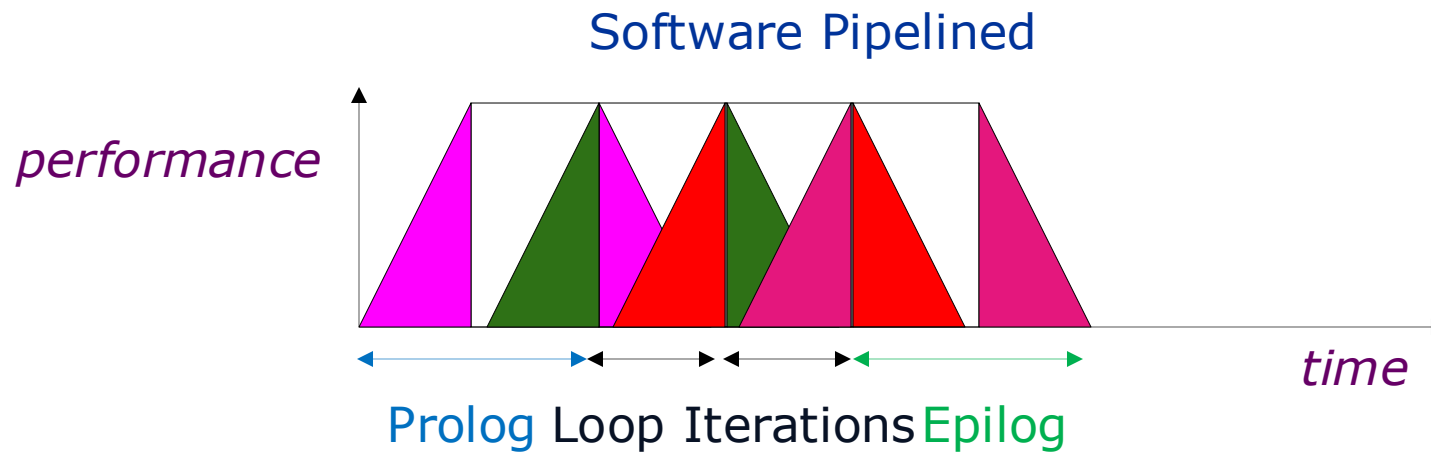
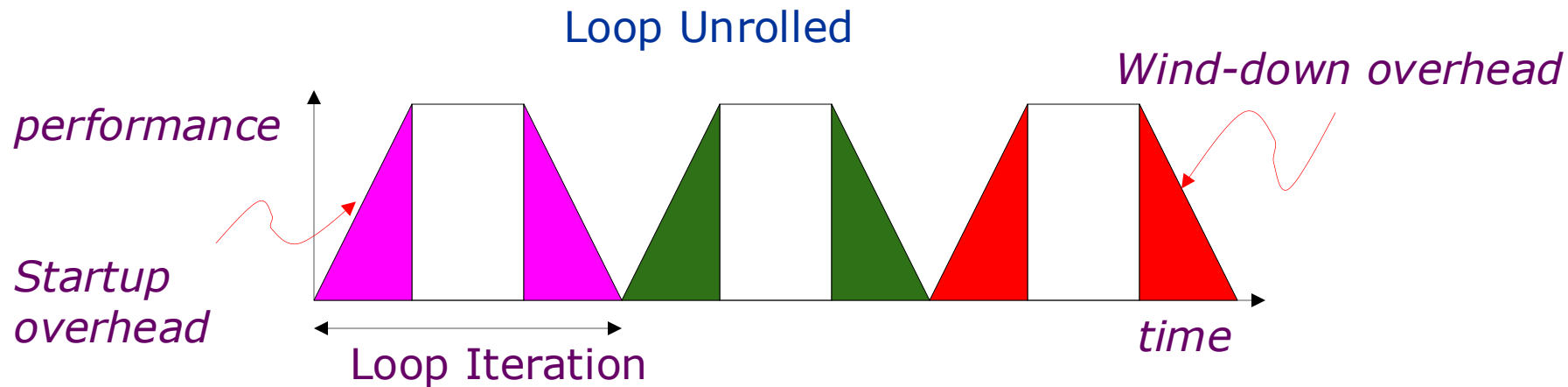
loop:
iterate

Epilog

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4			
		ld f1		fadd f5	
		ld f2		fadd f6	
		ld f3		fadd f7	
add r1		ld f4		fadd f8	
		ld f1	sd f5	fadd f5	
		ld f2	sd f6	fadd f6	
	add r2	ld f3	sd f7	fadd f7	
add r1	bne	ld f4	sd f8	fadd f8	
			sd f5	fadd f5	
			sd f6	fadd f6	
	add r2		sd f7	fadd f7	
	bne		sd f8	fadd f8	

sd f5

Loop Unrolling vs. Software Pipelining



Software pipelining pays startup/wind-down costs only once per loop, not once per iteration

--> the higher is the loop repetition, the less the prolog and epilog will count

Software Pipelining

Unroll 4 ways first

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
    
```

Prolog
Schedule

loop:
iterate

Epilog

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4			
		ld f1		fadd f5	
		ld f2		fadd f6	
		ld f3		fadd f7	
add r1		ld f4		fadd f8	
		ld f1	sd f5	fadd f5	
		ld f2	sd f6	fadd f6	
	add r2	ld f3	sd f7	fadd f7	
add r1	bne	ld f4	sd f8	fadd f8	
			sd f5	fadd f5	
			sd f6	fadd f6	
	add r2		sd f7	fadd f7	
	bne		sd f8	fadd f8	

sd f5

How many FLOPS/cycle?

Software Pipelining

Unroll 4 ways first

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
    
```

Prolog
Schedule

loop:
iterate

Epilog

Int1 1cc	Int 2 1cc	M1 3cc	M2 3cc	FP+ 4cc	FPx 4cc
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4			
		ld f1		fadd f5	
		ld f2		fadd f6	
		ld f3		fadd f7	
add r1		ld f4		fadd f8	
		ld f1	sd f5	fadd f5	
		ld f2	sd f6	fadd f6	
	add r2	ld f3	sd f7	fadd f7	
add r1	bne	ld f4	sd f8	fadd f8	
			sd f5	fadd f5	
			sd f6	fadd f6	
	add r2		sd f7	fadd f7	
	bne		sd f8	fadd f8	

How many FLOPS/cycle?

4 fadds / 4 cycles = 1

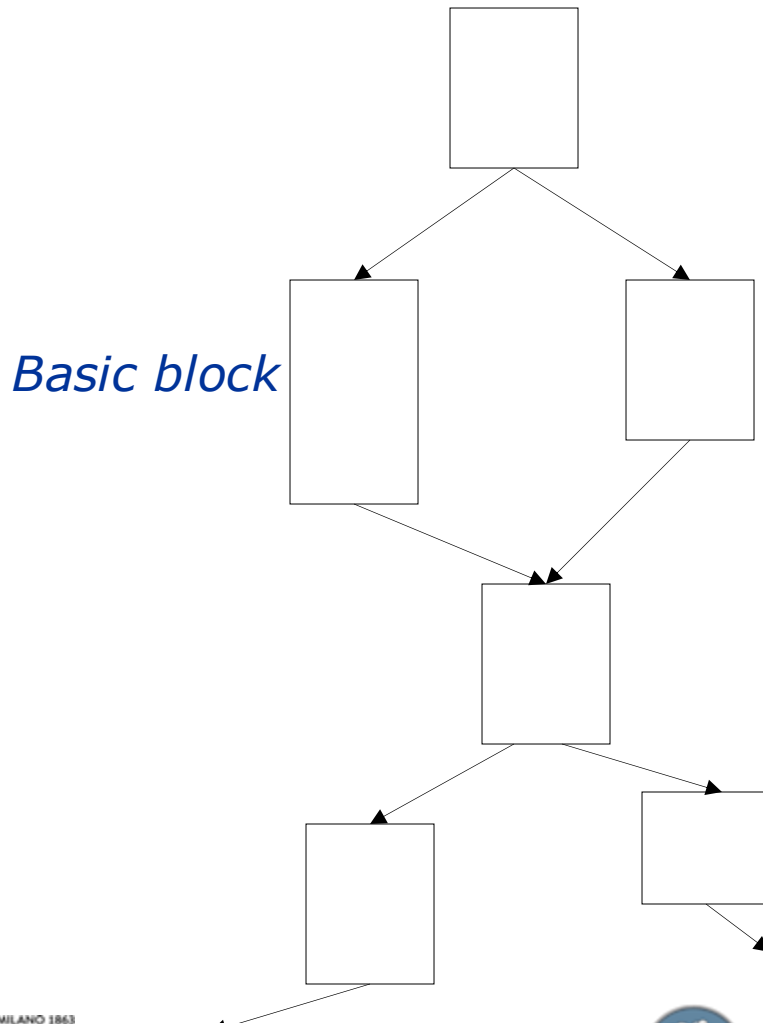
everything done by compiler!



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

What if there are no loops?



Branches limit basic block size in control-flow intensive irregular code

Difficult to find ILP in individual basic blocks

Trace scheduling: basic idea

- Trace scheduling focuses on **traces**
 - A trace is a **loop-free sequence** of basic blocks embedded in the control flow graph (Fisher)
 - It is an **execution path** which **can** be **taken** for some **set of inputs**
 - The chances that a trace is actually executed depends on the input set that allows its execution
- Some traces are executed much more frequently than others

Trace is a sequence of basic block. How to schedule instructions? Order traces. How? Profile reading: execute many times your code and measure how many times each schedule is going to be executed.

Pick the trace that with highest probability is going to be execute and schedule all the instruction in that trace. Then, if there are remaining traces to be scheduled, pick the one among them which has the highest probability to be executed and schedule ist instructions. And continue so on until traces are over.

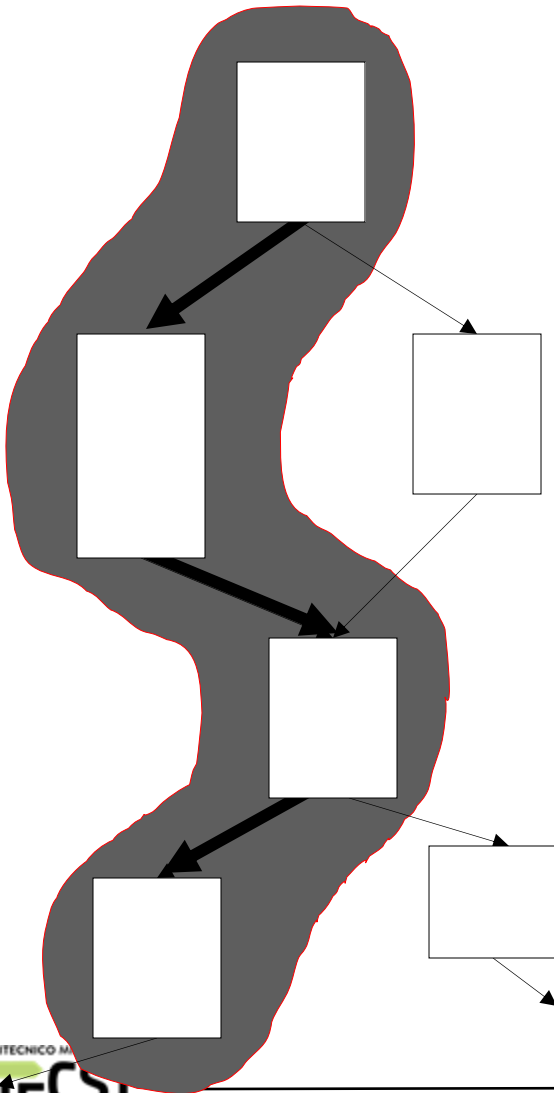
Trace Scheduling *[Fisher, Ellis]*

Pick string of basic blocks, a trace, that represents most frequent branch path

Use profiling feedback or compiler heuristics to find common branch paths

Schedule whole “trace” at once

Add fixup code to cope with branches jumping out of trace



Trace scheduling and loops

Trace scheduling and loops

- Trace scheduling **cannot** proceed **beyond a loop barrier**
- Techniques used to **overcome** this limitation are based on loop **unrolling**

Negative effects on unrolling

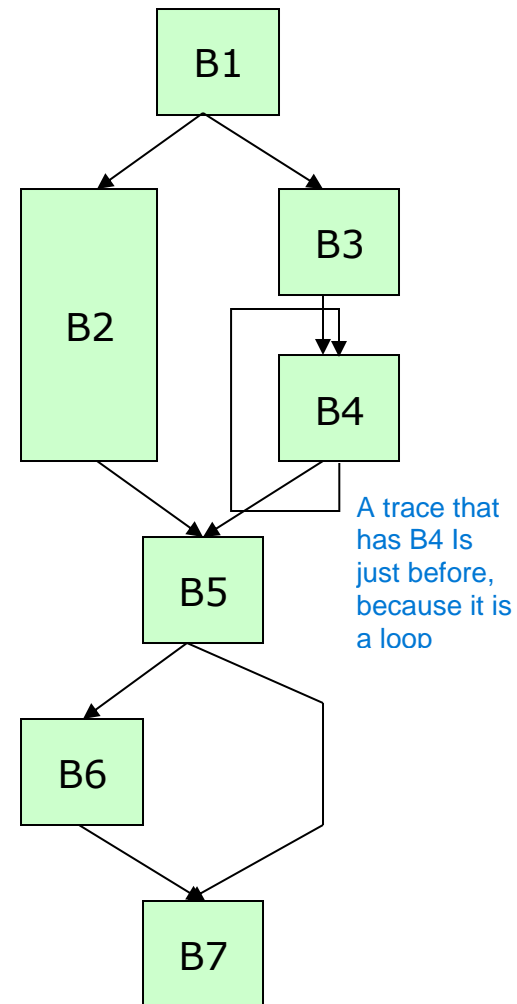
- **Unrolling** produces much **extra code**
- It also loses performance, because of the costs of starting and closing the iterations

Traces scheduling schedules traces in order of decreasing probability of being executed

- So, **most frequently executed traces get better schedules**
- Traces are **scheduled as if they were basic blocks** (no special considerations for branches)

Trace Scheduling: in deep

- **Trace** : a sequence of instructions which **may Include branches but not including loops**
- For example some traces in the control flow graph:
 - B1, B3
 - B4
 - B5, B7
 - B1, B2
 - B1, B2, B5, B6, B7



Trace Scheduling Cont'd

Trace Scheduling: finding **common path** and scheduling traces in that path independently

Scheduling in a trace rely on **basic code motion** but now has a global taste across more that one basic block by appropriate use of **renaming**

Compensation codes are needed

for side entry points: i.e. points except beginning
and slide exit points: i.e. points except ending

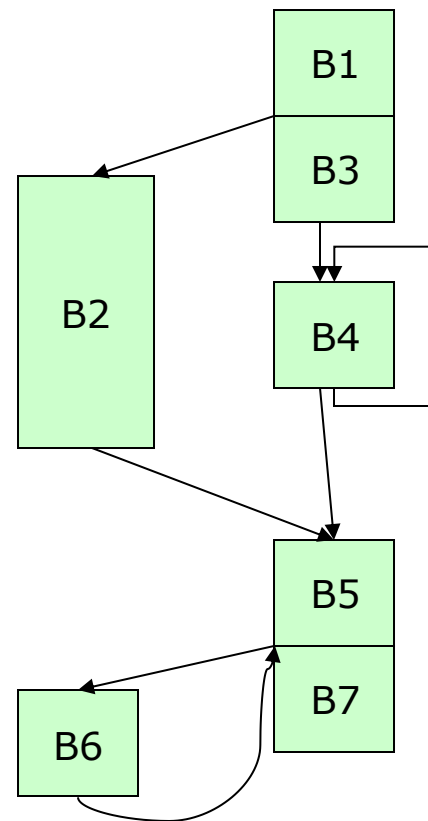
Blocks on non common path may now have added overhead, so there must be a high probability of taking common path according to profile (may not be clear for some programs)

Problems: **compensation codes are difficult to generate specially for entry points**

Trace Scheduling Example

For example suppose
that B1,B3,B4,B5,B7 is
the most frequently
executed path

Therefore traces are
B1,B3
B4 --> contained in a loop
B5,B7



Trace Scheduling Example

Using VLIW optimization inside a single basic block is not wise since there is a lot of waste of memory usage.

What if we are able to move code following us in order to schedule its instructions?

We need to go beyond edges between basic block in the same traces

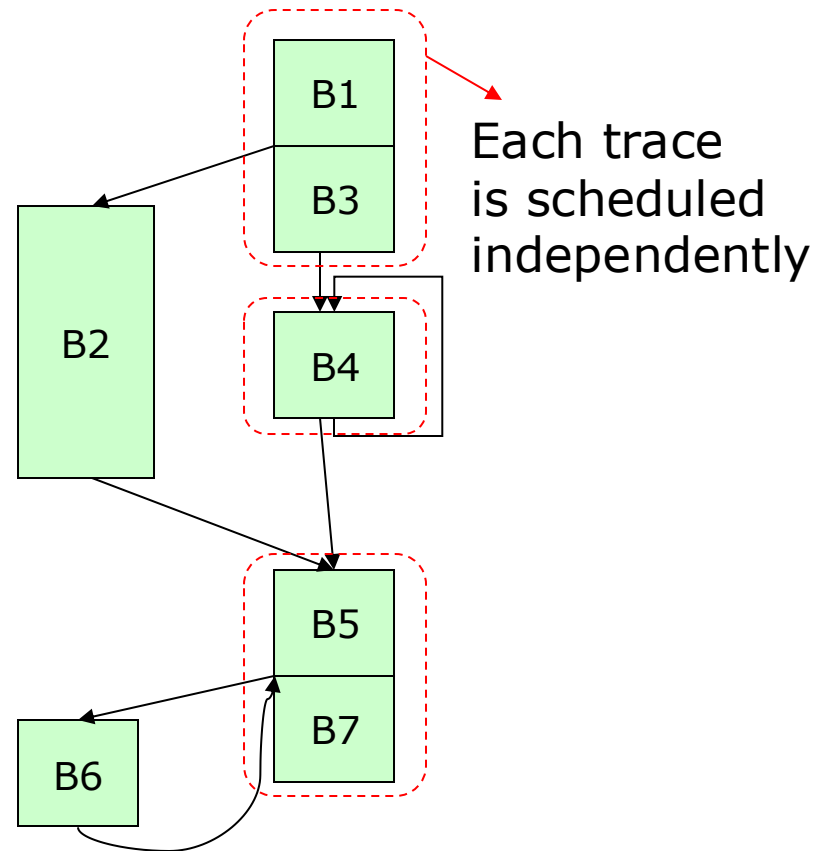
(es: border B1-B3) to achieve the best in performance. But doing so means that in order to fill empty slots in B1 instructions we look for code in B3.

What if we look for code in B2? This is a problem since that is not the most probable path. What if a runtime we actually have B1-B2 instead of B1-B3? We cannot guarantee correctness.

We will need to replicate code and refer to this as compensation code

For example suppose
that B1,B3,B4,B5,B7 is
the most frequently
executed path

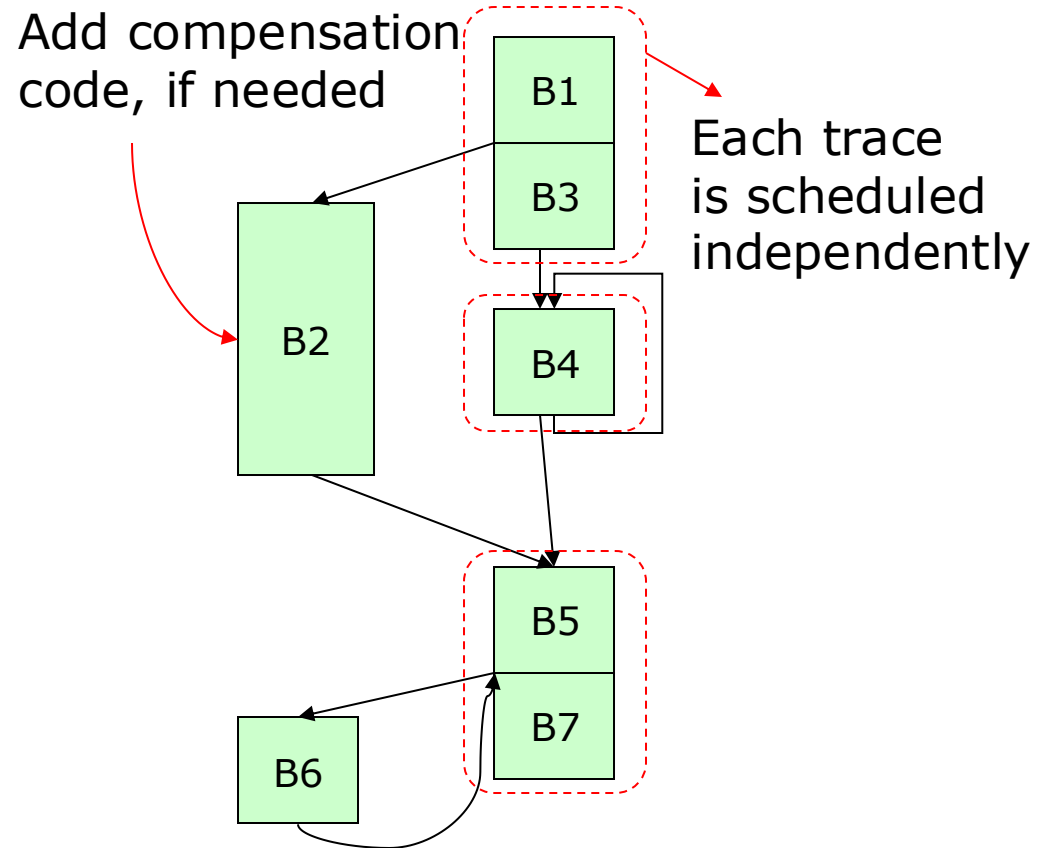
Therefore traces are
B1,B3
B4
B5,B7



Trace Scheduling Example

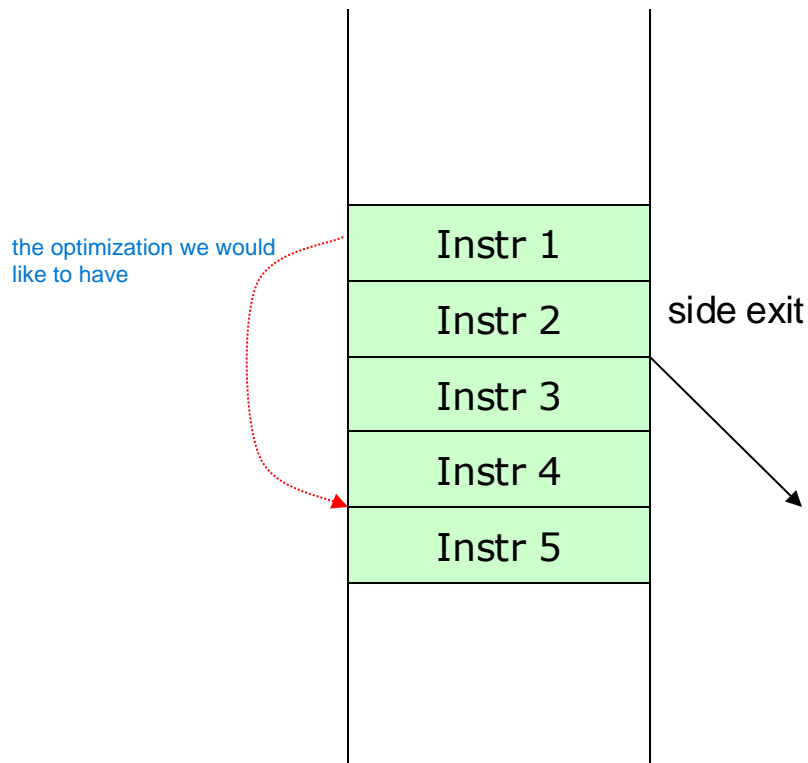
For example suppose
that B1,B3,B4,B5,B7 is
the most frequently
executed path

Therefore traces are
B1,B3
B4
B5,B7



Trace Scheduling Compensation 1

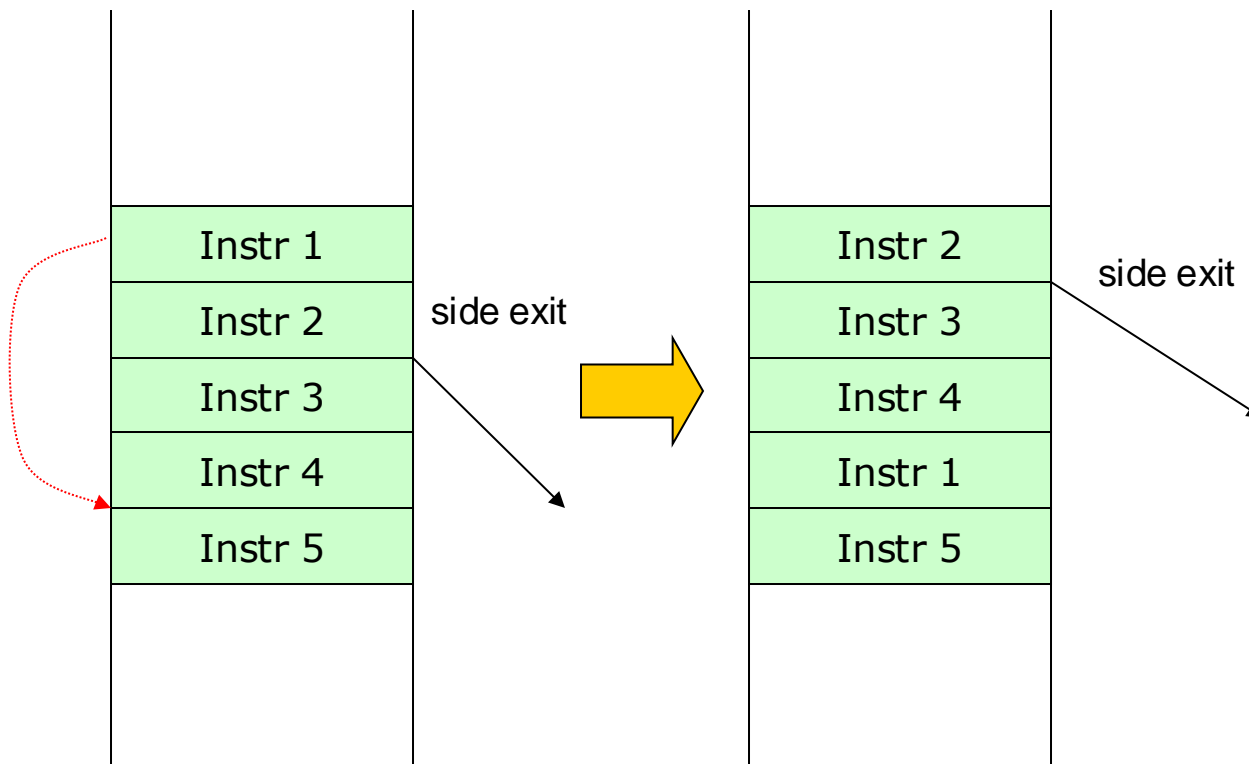
Moving an instruction below a side exit (simple)



Trace Scheduling Compensation 1

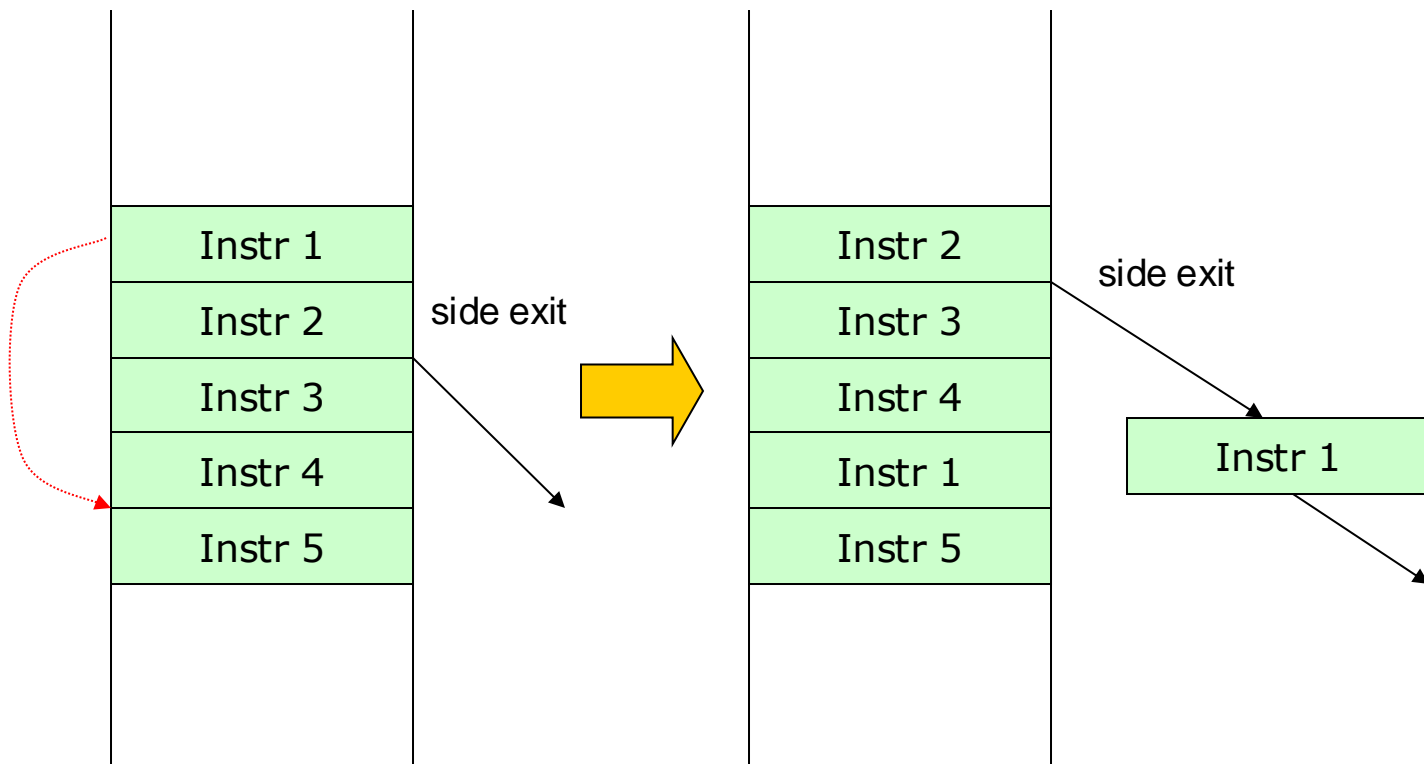
Moving an instruction below a side exit (simple)

We do the movement that was convenient



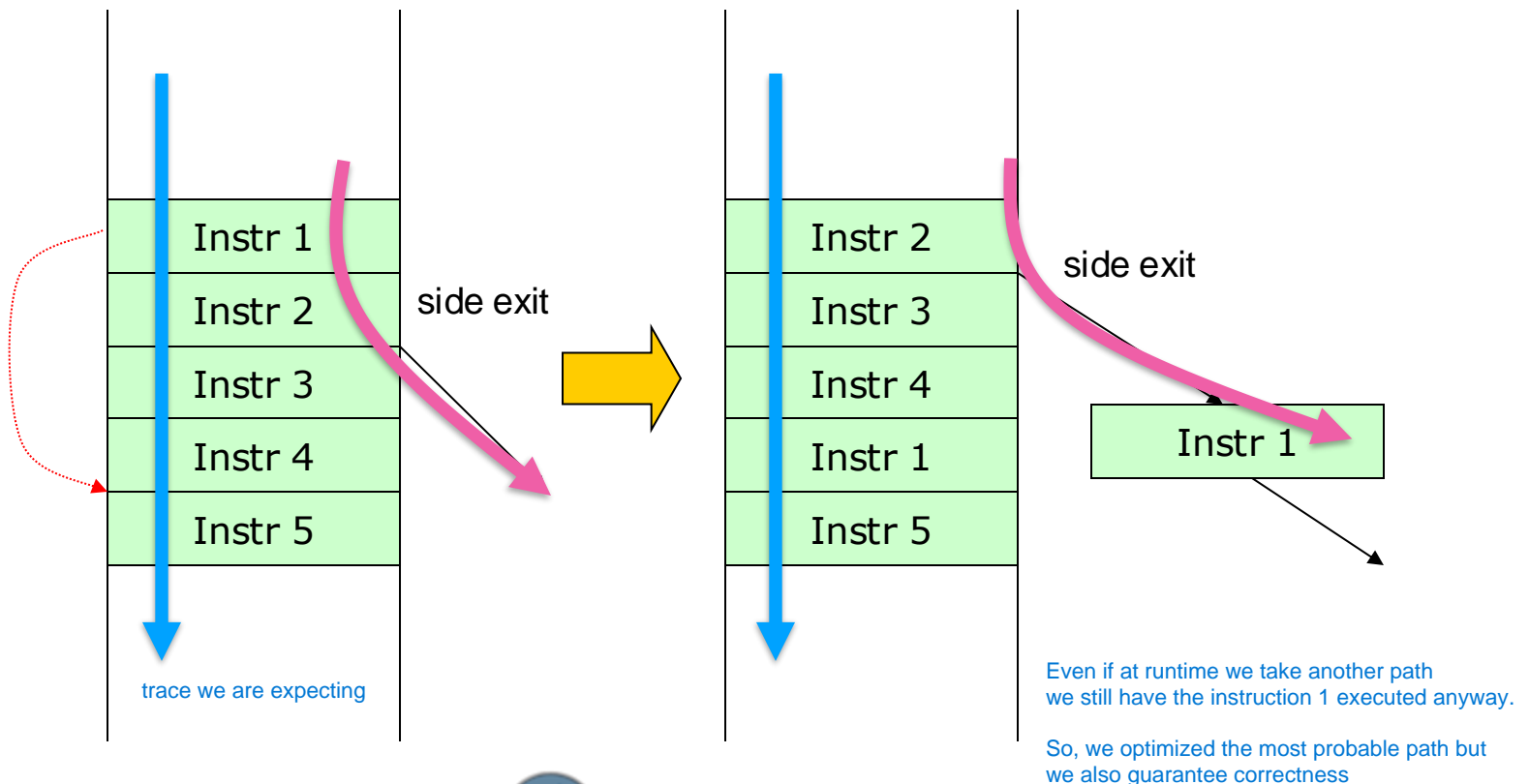
Trace Scheduling Compensation 1

Moving an instruction below a side exit (simple)



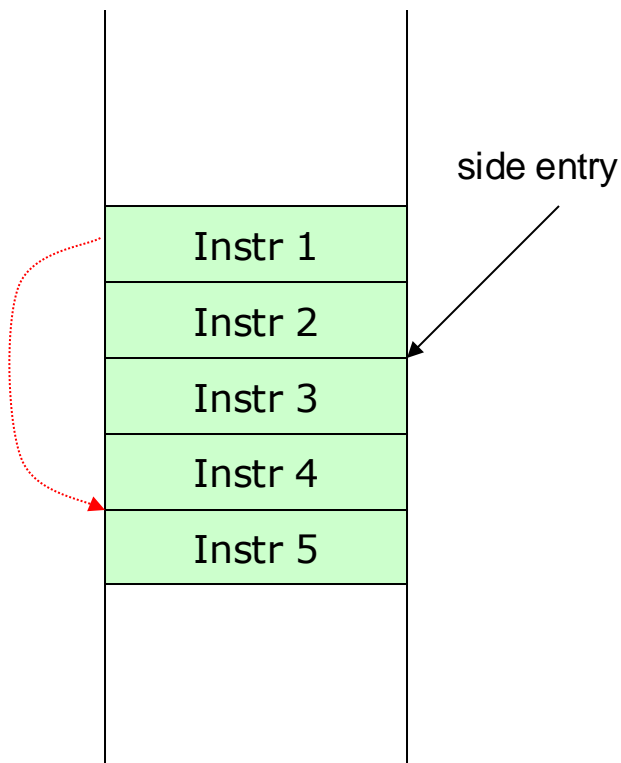
Trace Scheduling Compensation 1

Moving an instruction below a side exit (simple)



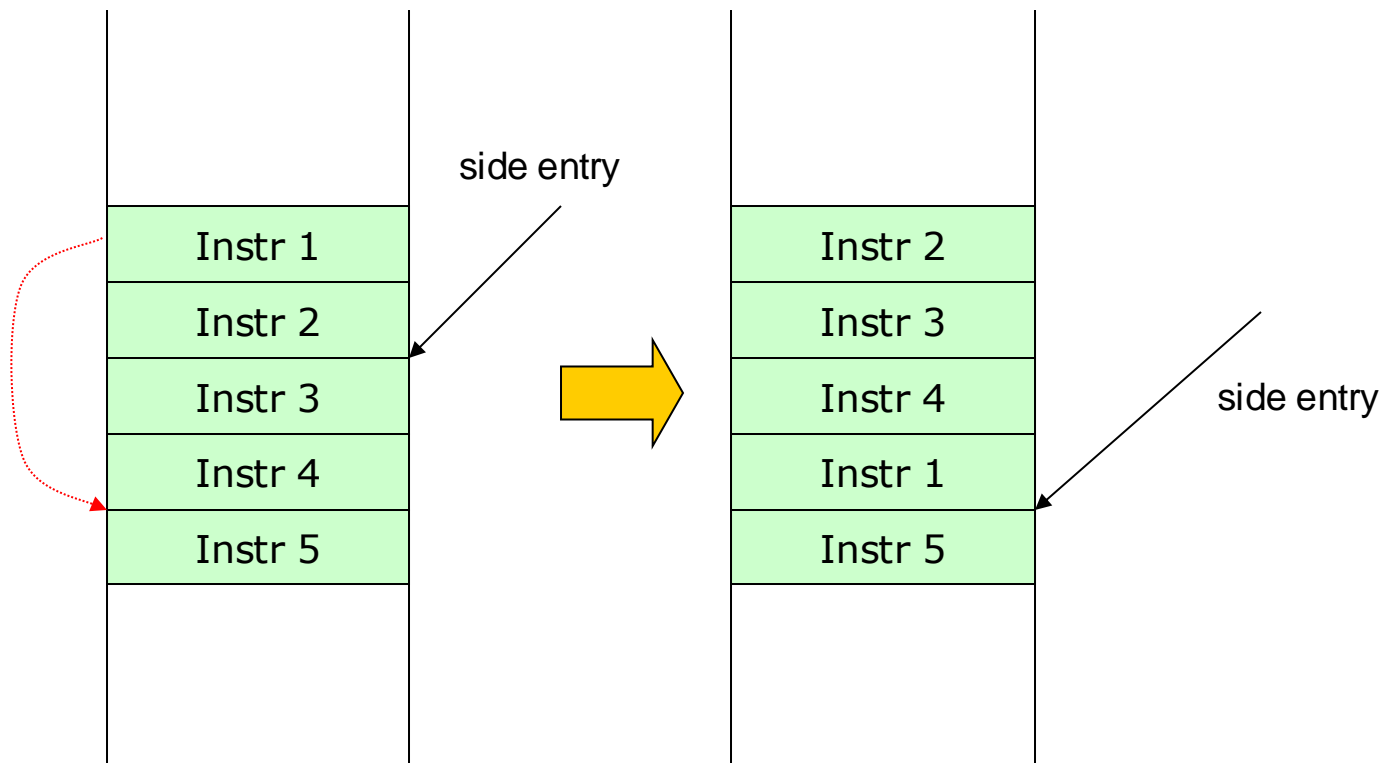
Trace Scheduling Compensation 2

Moving an instruction below a side entry (complex)



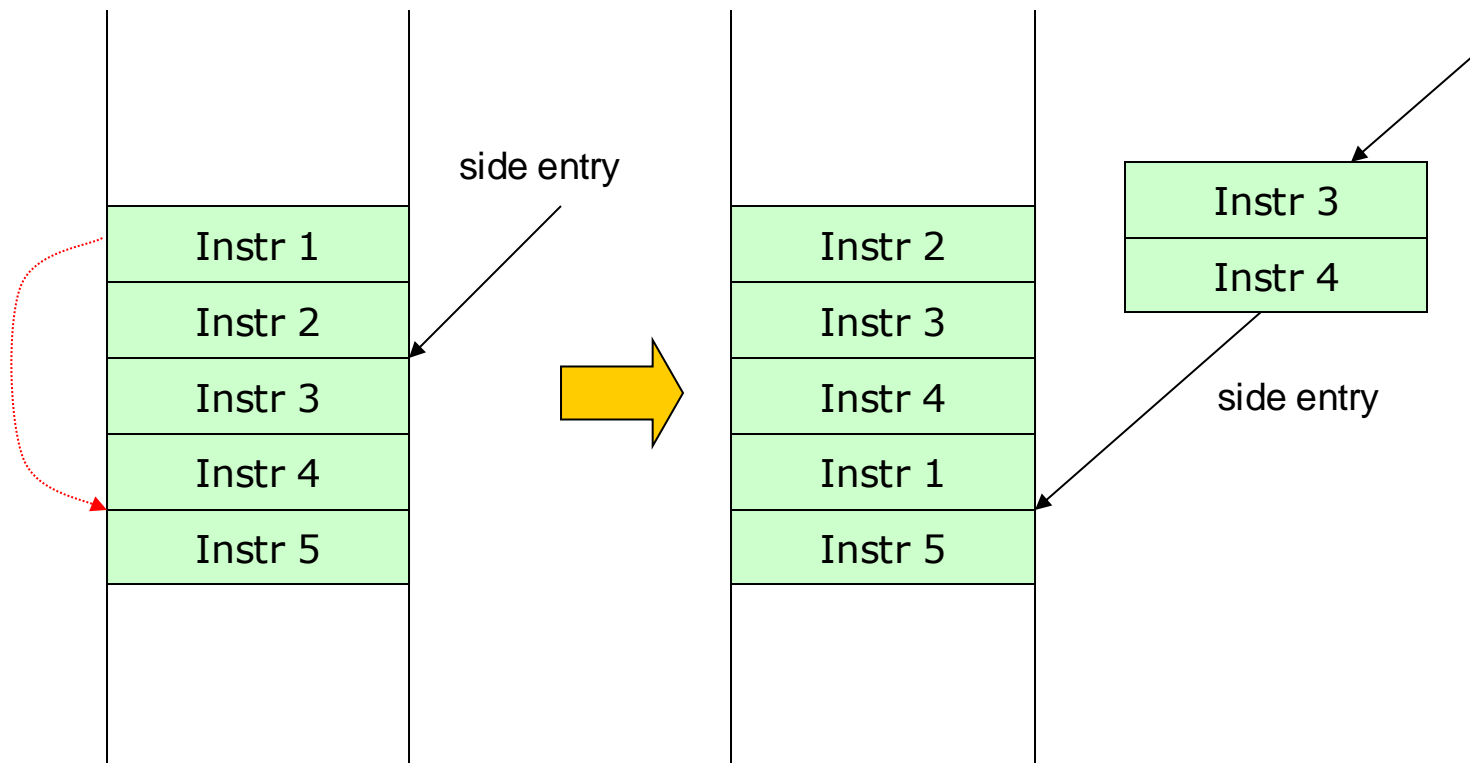
Trace Scheduling Compensation 2

Moving an instruction below a side entry (complex)



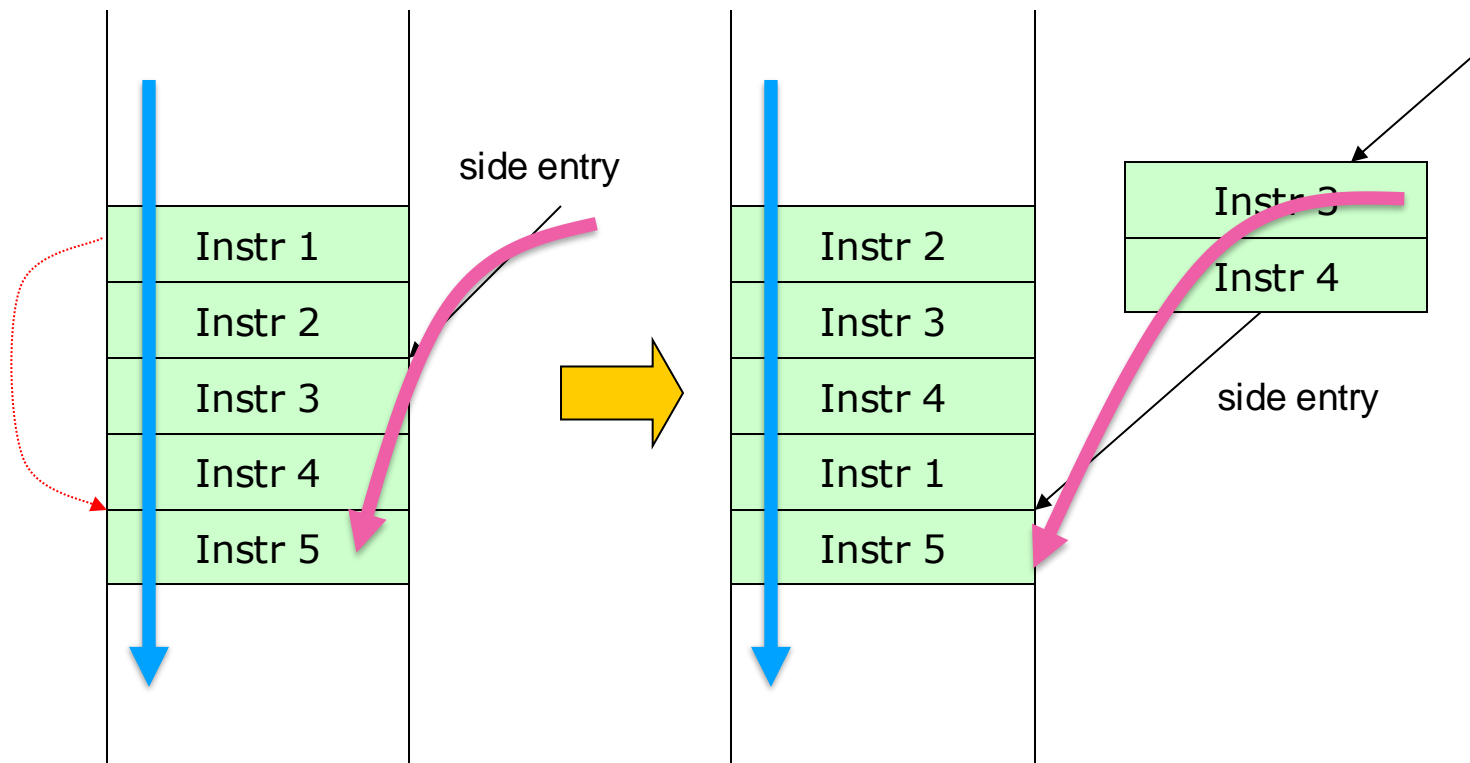
Trace Scheduling Compensation 2

Moving an instruction below a side entry (complex)



Trace Scheduling Compensation 2

Moving an instruction below a side entry (complex)



Code Motion in Trace Scheduling

In addition to need of compensation codes there are **restrictions on movement of a code in a trace:**

- The **dataflow** of the program must **not change**
- The **exception behavior** must be **preserved**

Dataflow can be guaranteed to be correct by **maintaining** two dependencies:

- **Data** dependency
- **Control** dependency

There are two solutions to eliminate control dependency:

- By use of predicate instructions (**Hyperblock scheduling**) and removing the branch.
- By use of speculative instructions (**Speculative Scheduling**) and speculatively move an instruction before the branch.

Code Motion in Trace Scheduling

In addition to need of compensation codes there are restrictions on movement of a code in a trace:

- The dataflow of the program must not change
- The exception behavior must be preserved

Dataflow can be guaranteed to be correct by **maintaining** two dependencies:

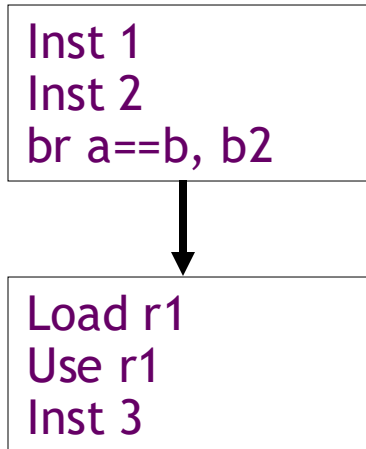
- **Data** dependency
- **Control** dependency

There are two solutions to eliminate control dependency:

- By use of predicate instructions (Hyperblock scheduling) and removing the branch.
- By use of speculative instructions (Speculative Scheduling) and speculatively move an instruction before the branch.

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don't cause exceptions



*Can't move load above branch
because might cause spurious
exception*

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don't cause exceptions

Inst 1
Inst 2
br a==b, b2

Load r1
Use r1
Inst 3

*Can't move load above branch
because might cause spurious
exception*

Load.s r1
Inst 1
Inst 2
br a==b, b2

Chk.s r1
Use r1
Inst 3

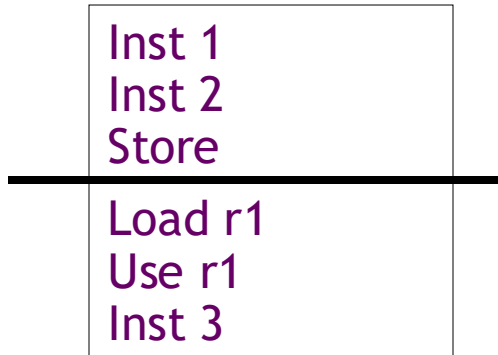
*Speculative load
never causes
exception, but
sets "poison" bit
on destination
register*

*Check for exception
in original home
block jumps to fixup
code if exception
detected*

Particularly useful for scheduling long latency loads early

Problem: Possible memory hazards limit code scheduling

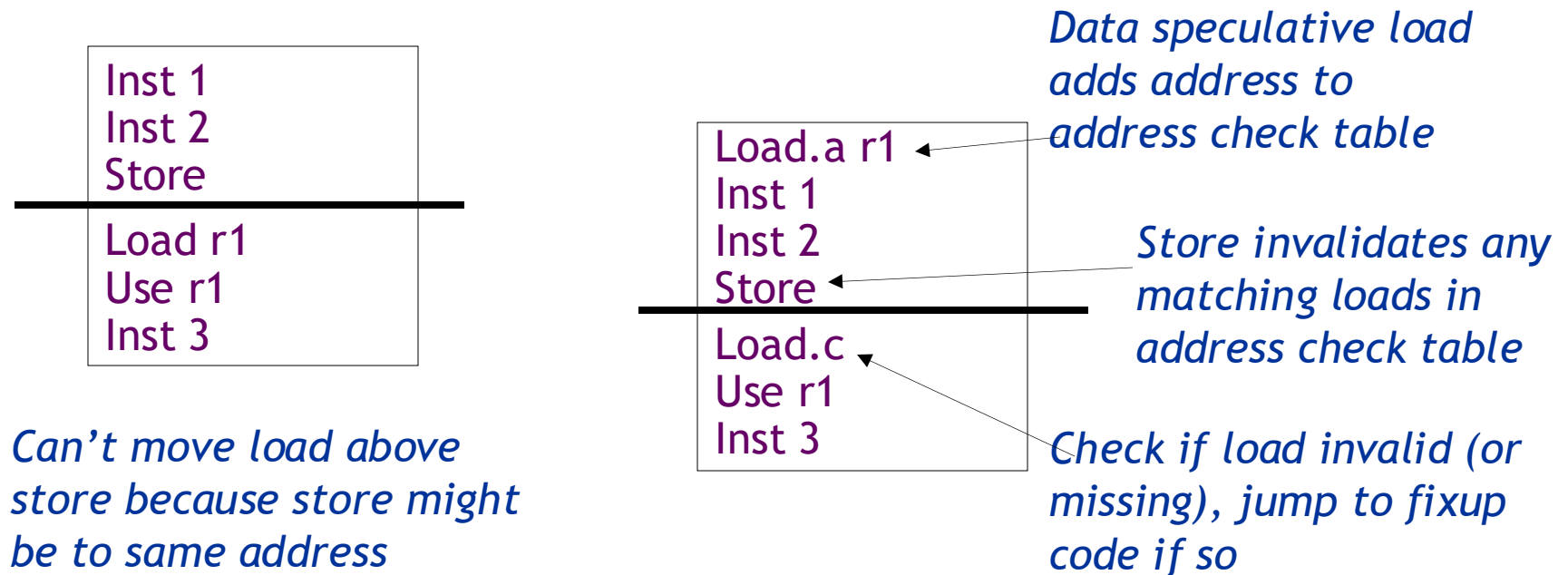
Solution: Hardware to check pointer hazards



*Can't move load above
store because store might
be to same address*

Problem: Possible memory hazards limit code scheduling

Solution: Hardware to check pointer hazards



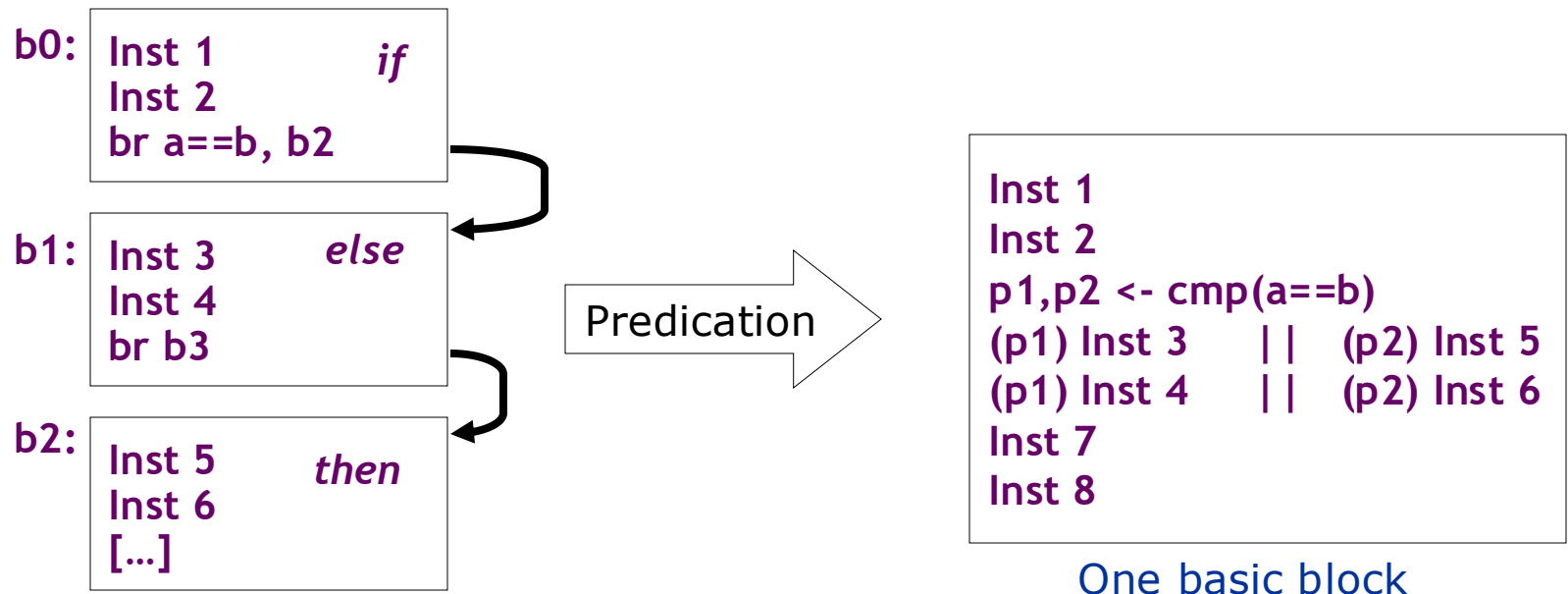
Requires associative hardware in address check table

Predicated Execution

Problem: Mispredicted branches limit ILP

Solution: Eliminate hard to predict branches with predicated execution

- Almost all IA-64 instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false



Thank you for your attention

Questions?

Davide Conficconi <davide.conficconi@polimi.it>

Acknowledgements

Marco D. Santambrogio, D. Sciuto and their previous credits

Part of this material comes from:

- Hennessy and Patterson [Turing Lecture](#)
- “Computer Organization and Design” and “Computer Architecture A Quantitative Approach” Patterson and Hennessy books
- Elsevier Inc. online materials
- Paper & News cited in the lecture

and are **properties of their respective owners**

!!!EXTRA!!!

FINISH LINE IN SIGHT



Rotating Register Files

Problems:

Scheduled loops require lots of registers,
Lots of duplicated code in prolog, epilog

Rotating Register Files

Problems:

Scheduled loops require lots of registers,
Lots of duplicated code in prolog, epilog

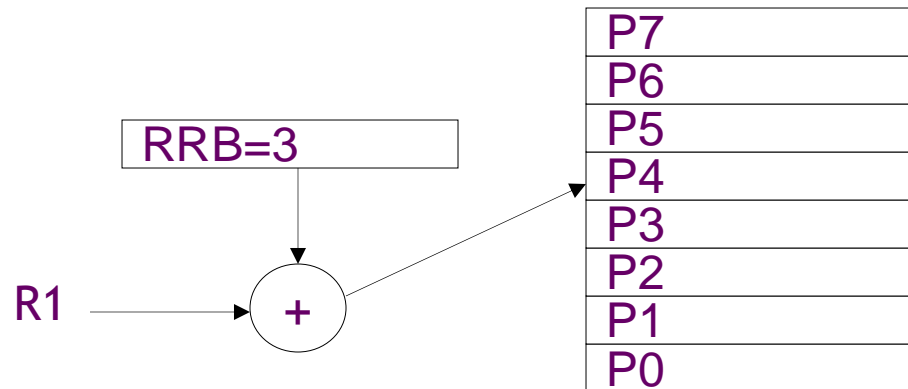
Solution:

Allocate new set of registers for each loop
iteration

Rotating Register File

Rotating Register Base (RRB) register points to base of current register set.

Value added on to logical register specifier to give physical register number.



Usually, split into rotating and non-rotating registers.

Loop Example

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

Compile

```
loop: ld f1, 0(r1)  
      add r1, 8  
      fadd f2, f0, f1  
      sd f2, 0(r2)  
      add r2, 8  
      bne r1, r3, loop
```

loop:

Schedule

Int1	Int 2	M1	M2	FP+	FPx
add r1		ld			
				fadd	
add r2	bne		sd		

Loop example: what we'd like to...

Int1 Int 2 M1 M2 FP+ FPx

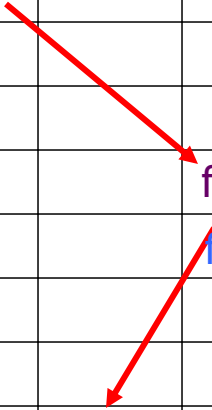
add r1		ld			
				fadd	
add r2	bne		sd		

No dep on FP in consecutive iterations

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

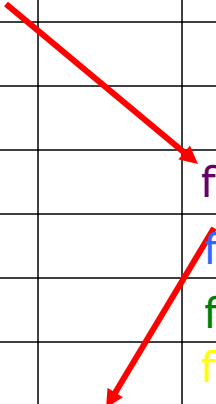
Loop example: what we'd like to...

Int1	Int 2	M1	M2	FP+	FPx
add r1		ld			
		ld			
				fadd	
				fadd	
add r2	bne		sd		
			sd		



Loop example: what we'd like to...

Int1	Int 2	M1	M2	FP+	FPx
add r1		ld			
		ld			
		ld			
		ld		fadd	
		ld		fadd	
		ld		fadd	
		ld		fadd	
add r2	bne	ld	sd	fadd	
			sd	fadd	



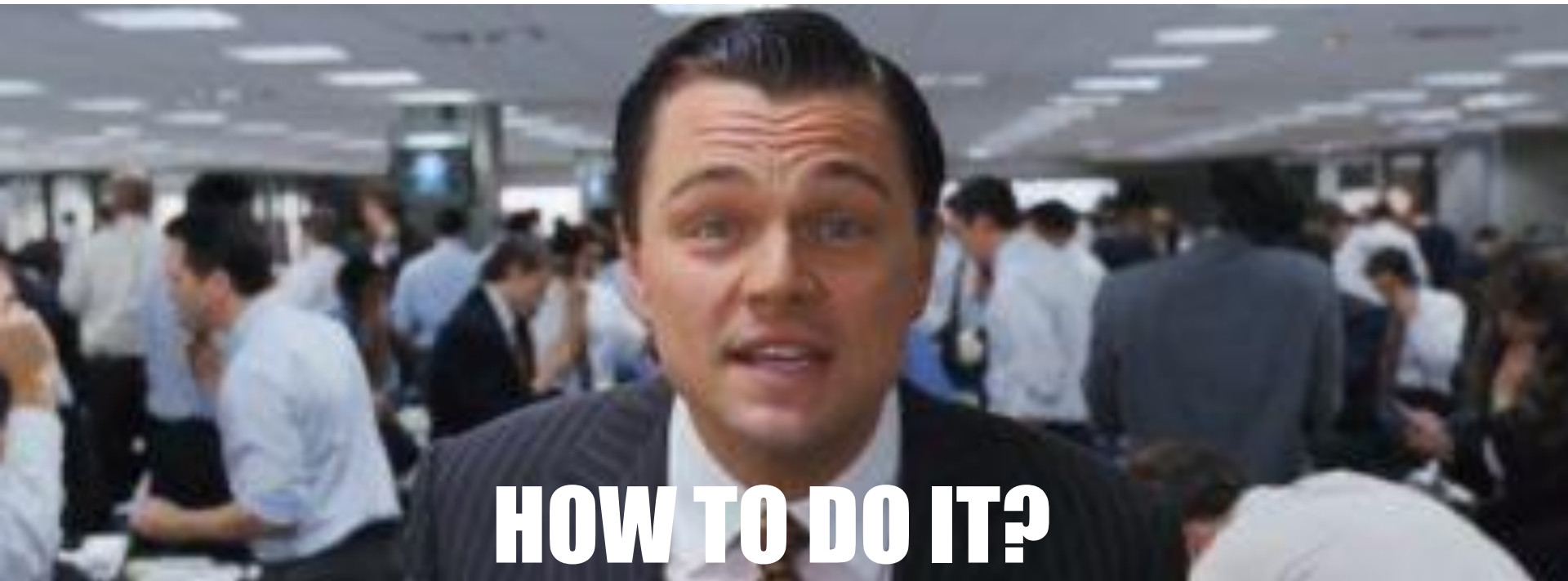
Loop example: what we'd like to...

	Int1	Int 2	M1	M2	FP+	FPx
add r1			ld			
			ld			
			ld			
			ld		fadd	
			ld		fadd	
			ld		fadd	
			ld		fadd	
add r2	bne		ld	sd	fadd	
				sd	fadd	

ld f1, ()	fadd fx, fy, ...	sd fz, ()	bloop
-----------	------------------	-----------	-------

Loop example: what we'd like to...

Int1 Int 2 M1 M2 FP+ FPx



HOW TO DO IT?

ld f1, ()

fadd fx, fy, ...

sd fz, ()

bloop

Rotating Register File

ld f1, ()	fadd fx, fy, ...	sd fz, ()	bloop
-----------	------------------	-----------	-------

Rotating Register File

ld f1, ()	fadd fx, fy, ...	sd fz, ()	bloop
-----------	------------------	-----------	-------

Three cycle load latency
encoded as difference of 3 in
register specifier number

Four cycle fadd latency
encoded as difference of 4 in
register specifier number

Rotating Register File

ld f1, ()	fadd fx, fy, ...	sd fz, ()	bloop
-----------	------------------	-----------	-------

Three cycle load latency
encoded as difference of 3 in
register specifier number
($f1 + 3 = fy \dots y = 3+1$)

Four cycle fadd latency
encoded as difference of 4 in
register specifier number
($f5 + 4 = fz \dots z = 5+4$)

ld f1, ()	fadd f5, f4, ...	sd f9, ()	bloop
-----------	------------------	-----------	-------

```
loop: ld f1, ...  
      ...  
      fadd f2, f0, f1  
      sd f2, ...
```

Rotating Register File

Three cycle load latency
encoded as difference of 3 in
register specifier number
($f1 + 3 = fy \dots y = 3+1$)

Four cycle fadd latency
encoded as difference of 4 in
register specifier number
($f5 + 4 = fz \dots z = 5+4$)



Rotating Register File

Three cycle load latency
encoded as difference of 3 in
register specifier number
($f1 + 3 = fy \dots y = 3+1$)

Four cycle fadd latency
encoded as difference of 4 in
register specifier number
($f5 + 4 = fz \dots z = 5+4$)

ld f1, ()	fadd f5, f4, ...	sd f9, ()	bloop
ld P9, ()	fadd P13, P12,	sd P17, ()	bloop

RRB=8

Rotating Register File

Three cycle load latency
encoded as difference of 3 in
register specifier number
($f1 + 3 = fy \dots y = 3+1$)

Four cycle fadd latency
encoded as difference of 4 in
register specifier number
($f5 + 4 = fz \dots z = 5+4$)

ld f1, ()	fadd f5, f4, ...	sd f9, ()	bloop
ld P9, ()	fadd P13, P12,	sd P17, ()	bloop
ld P8, ()	fadd P12, P11,	sd P16, ()	bloop

RRB=8

RRB=7

Rotating Register File

Three cycle load latency
encoded as difference of 3 in
register specifier number
($f1 + 3 = fy \dots y = 3+1$)

Four cycle fadd latency
encoded as difference of 4 in
register specifier number
($f5 + 4 = fz \dots z = 5+4$)

ld f1, ()	fadd f5, f4, ...	sd f9, ()	bloop
ld P9, ()	fadd P13, P12,	sd P17, ()	bloop
ld P8, ()	fadd P12, P11,	sd P16, ()	bloop
ld P7, ()	fadd P11, P10,	sd P15, ()	bloop

RRB=8

RRB=7

RRB=6

Rotating Register File

Three cycle load latency
encoded as difference of 3 in
register specifier number
($f1 + 3 = fy \dots y = 3+1$)

Four cycle fadd latency
encoded as difference of 4 in
register specifier number
($f5 + 4 = fz \dots z = 5+4$)

ld f1, ()	fadd f5, f4, ...	sd f9, ()	bloop
ld P9, ()	fadd P13, P12,	sd P17, ()	bloop
ld P8, ()	fadd P12, P11,	sd P16, ()	bloop
ld P7, ()	fadd P11, P10,	sd P15, ()	bloop
ld P6, ()	fadd P10, P9,	sd P14, ()	bloop

RRB=8

RRB=7

RRB=6

RRB=5

Rotating Register File

Three cycle load latency
encoded as difference of 3 in
register specifier number
($f1 + 3 = fy \dots y = 3+1$)

Four cycle fadd latency
encoded as difference of 4 in
register specifier number
($f5 + 4 = fz \dots z = 5+4$)

ld f1, ()	fadd f5, f4, ...	sd f9, ()	bloop
ld P9, ()	fadd P13, P12,	sd P17, ()	bloop
ld P8, ()	fadd P12, P11,	sd P16, ()	bloop
ld P7, ()	fadd P11, P10,	sd P15, ()	bloop
ld P6, ()	fadd P10, P9,	sd P14, ()	bloop

RRB=8

RRB=7

RRB=6

RRB=5

Rotating Register File

Three cycle load latency
encoded as difference of 3 in
register specifier number
($f1 + 3 = fy \dots y = 3+1$)

Four cycle fadd latency
encoded as difference of 4 in
register specifier number
($f5 + 4 = fz \dots z = 5+4$)

ld f1, ()	fadd f5, f4, ...	sd f9, ()	bloop
ld P9, ()	fadd P13, P12,	sd P17, ()	bloop
ld P8, ()	fadd P12, P11,	sd P16, ()	bloop
ld P7, ()	fadd P11, P10,	sd P15, ()	bloop
ld P6, ()	fadd P10, P9,	sd P14, ()	bloop
ld P5, ()	fadd P9, P8,	sd P13, ()	bloop

RRB=8

RRB=7

RRB=6

RRB=5

RRB=4

Rotating Register File

Three cycle load latency
encoded as difference of 3 in
register specifier number
($f1 + 3 = fy \dots y = 3+1$)

Four cycle fadd latency
encoded as difference of 4 in
register specifier number
($f5 + 4 = fz \dots z = 5+4$)

ld f1, ()	fadd f5, f4, ...	sd f9, ()	bloop
ld P9, ()	fadd P13, P12,	sd P17, ()	bloop
ld P8, ()	fadd P12, P11,	sd P16, ()	bloop
ld P7, ()	fadd P11, P10,	sd P15, ()	bloop
ld P6, ()	fadd P10, P9,	sd P14, ()	bloop
ld P5, ()	fadd P9, P8,	sd P13, ()	bloop
ld P4, ()	fadd P8, P7,	sd P12, ()	bloop

RRB=8

RRB=7

RRB=6

RRB=5

RRB=4

RRB=3

Rotating Register File

Three cycle load latency
encoded as difference of 3 in
register specifier number
($f1 + 3 = fy \dots y = 3+1$)

Four cycle fadd latency
encoded as difference of 4 in
register specifier number
($f5 + 4 = fz \dots z = 5+4$)

ld f1, ()	fadd f5, f4, ...	sd f9, ()	bloop
ld P9, ()	fadd P13, P12,	sd P17, ()	bloop
ld P8, ()	fadd P12, P11,	sd P16, ()	bloop
ld P7, ()	fadd P11, P10,	sd P15, ()	bloop
ld P6, ()	fadd P10, P9,	sd P14, ()	bloop
ld P5, ()	fadd P9, P8,	sd P13, ()	bloop
ld P4, ()	fadd P8, P7,	sd P12, ()	bloop
ld P3, ()	fadd P7, P6,	sd P11, ()	bloop

RRB=8

RRB=7

RRB=6

RRB=5

RRB=4

RRB=3

RRB=2

Rotating Register File

Three cycle load latency
encoded as difference of 3 in
register specifier number
($f1 + 3 = fy \dots y = 3+1$)

Four cycle fadd latency
encoded as difference of 4 in
register specifier number
($f5 + 4 = fz \dots z = 5+4$)

ld f1, ()	fadd f5, f4, ...	sd f9, ()	bloop
ld P9, ()	fadd P13, P12,	sd P17, ()	bloop
ld P8, ()	fadd P12, P11,	sd P16, ()	bloop
ld P7, ()	fadd P11, P10,	sd P15, ()	bloop
ld P6, ()	fadd P10, P9,	sd P14, ()	bloop
ld P5, ()	fadd P9, P8,	sd P13, ()	bloop
ld P4, ()	fadd P8, P7,	sd P12, ()	bloop
ld P3, ()	fadd P7, P6,	sd P11, ()	bloop
ld P2, ()	fadd P6, P5,	sd P10, ()	bloop

RRB=8

RRB=7

RRB=6

RRB=5

RRB=4

RRB=3

RRB=2

RRB=1

Rotating Register File

Three cycle load latency
encoded as difference of 3 in
register specifier number
($f1 + 3 = fy \dots y = 3+1$)

Four cycle fadd latency
encoded as difference of 4 in
register specifier number
($f5 + 4 = fz \dots z = 5+4$)

ld f1, ()	fadd f5, f4, ...	sd f9, ()	bloop
ld P9, ()	fadd P13, P12,	sd P17, ()	bloop
ld P8, ()	fadd P12, P11,	sd P16, ()	bloop
ld P7, ()	fadd P11, P10,	sd P15, ()	bloop
ld P6, ()	fadd P10, P9,	sd P14, ()	bloop
ld P5, ()	fadd P9, P8,	sd P13, ()	bloop
ld P4, ()	fadd P8, P7,	sd P12, ()	bloop
ld P3, ()	fadd P7, P6,	sd P11, ()	bloop
ld P2, ()	fadd P6, P5,	sd P10, ()	bloop

RRB=8

RRB=7

RRB=6

RRB=5

RRB=4

RRB=3

RRB=2

RRB=1

END...

