Advanced Computer Architecture

—

Part I: General Purpose
Exploiting ILP Statically

Paolo.Ienne@epfl.ch

EPFL – I&C – LAP

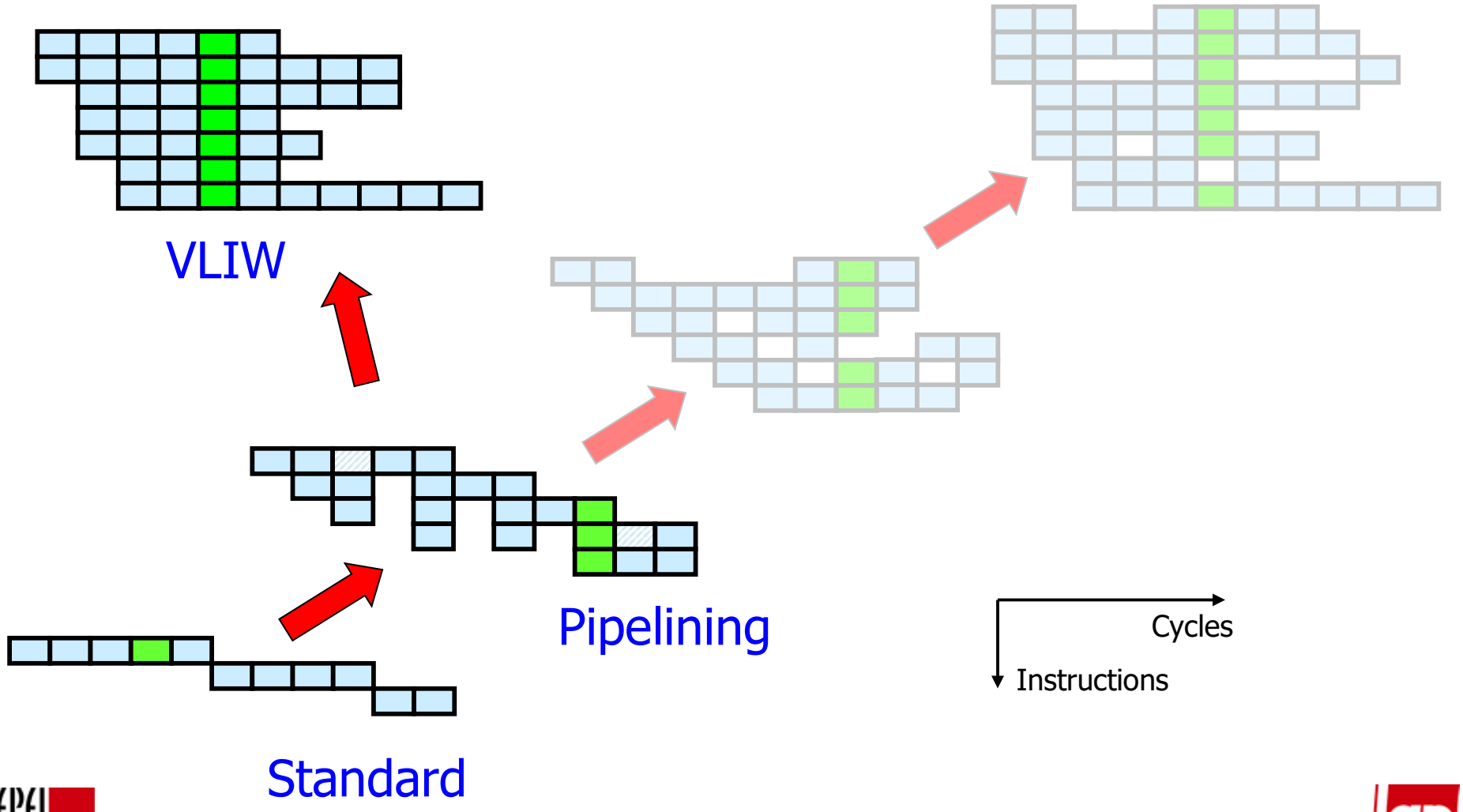
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

1

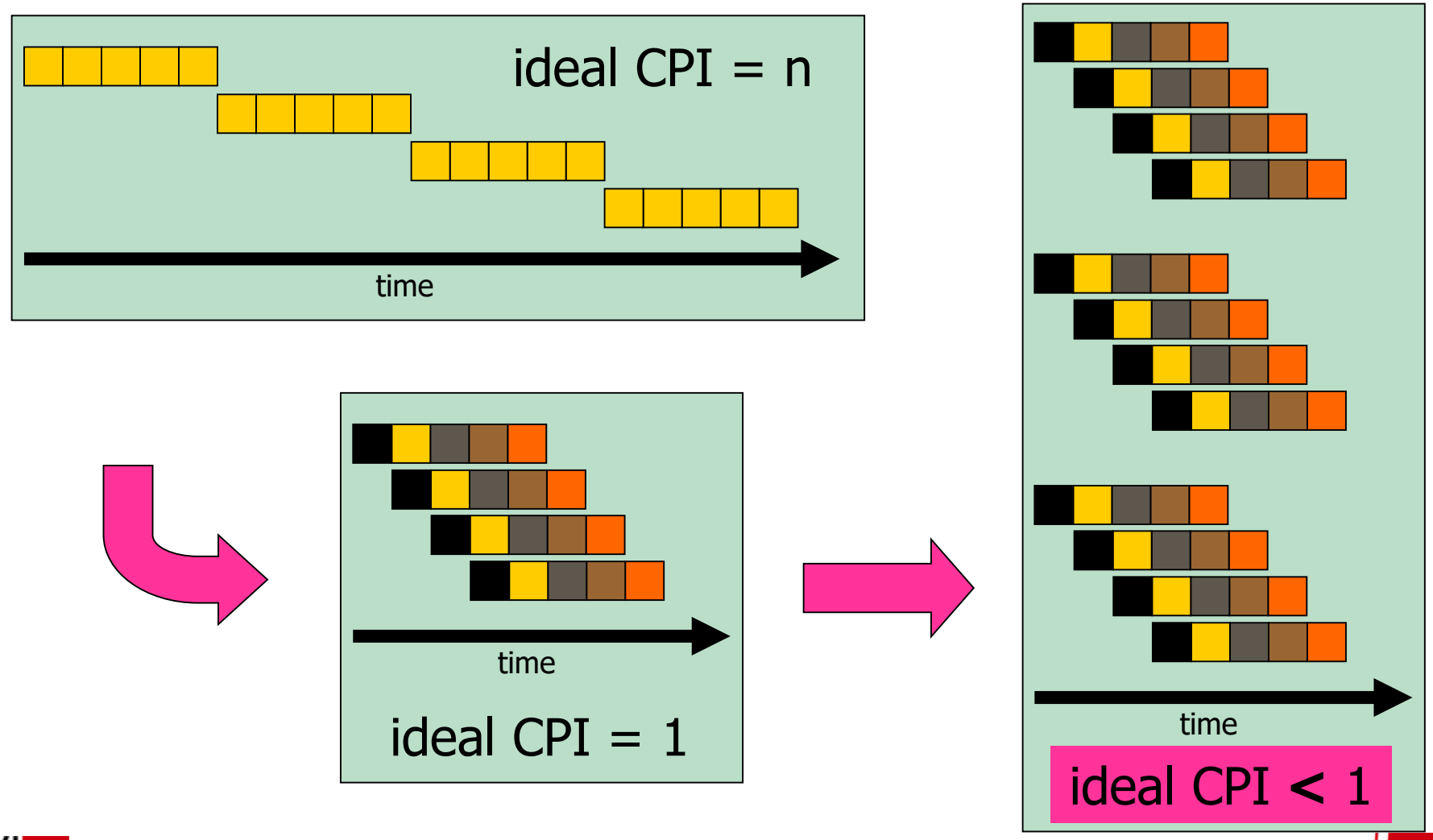
VLIW and EPIC? Another Way to ILP

(What if I Now Threw It All Away?!...)

Very Long Instruction Word: An Alternate Way of Extracting ILP



Sequential → Pipelined → Multiple Issue



3 Requirements to Obtain CPI < 1

1. **Machine parallelism**

   The machine is equipped with multiple datapaths (pipelines)

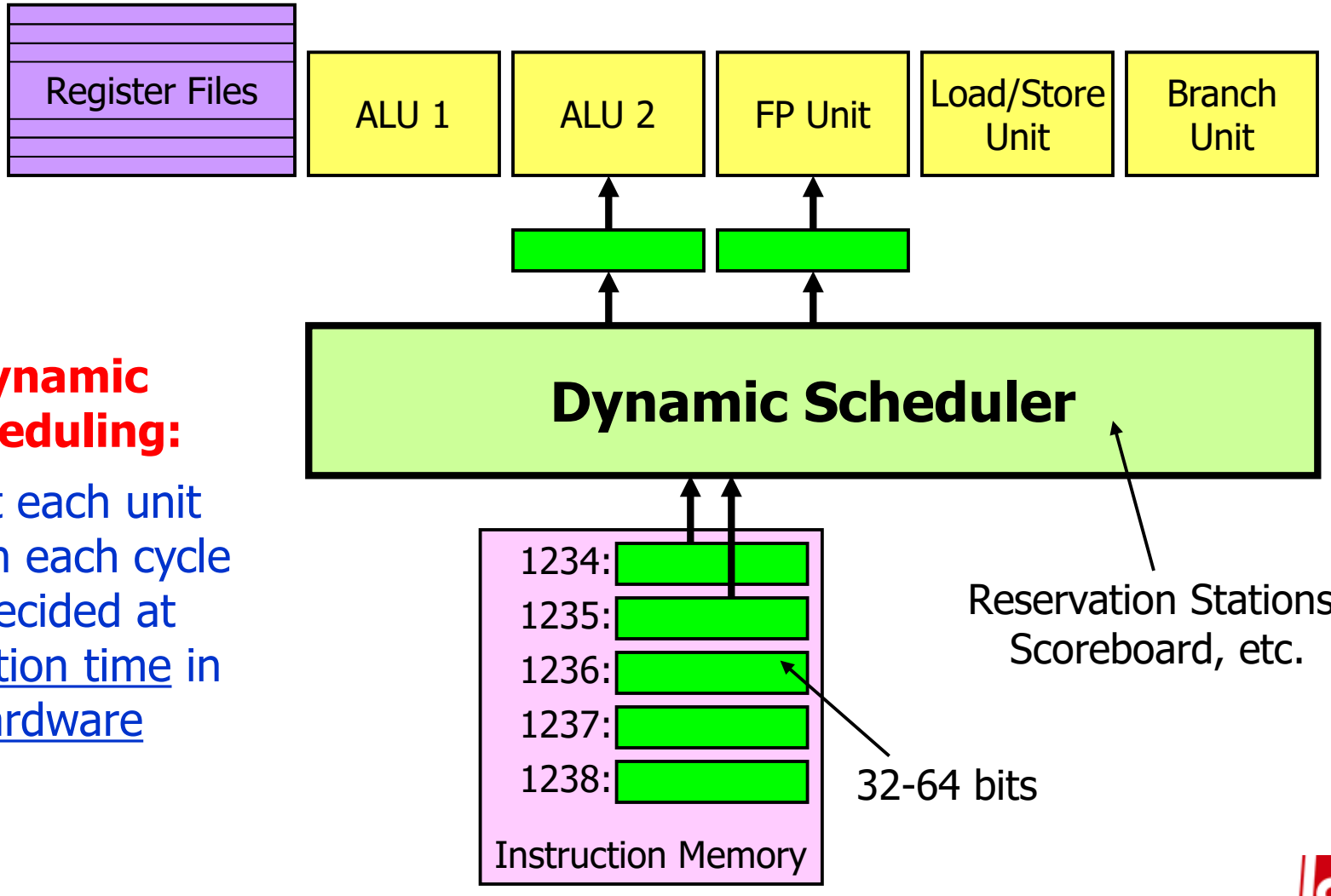
2. **Application parallelism**

   The application program has inherent parallelism that can be exploited

3. **Compiler "cleverness"**

   The compiler needs to **discover** the application parallelism and **expose it** to the machine

(Dynamically Scheduled) Superscalar Processor



Dynamic Scheduling:

What each unit does in each cycle is decided at execution time in hardware

Run Time vs. Compiler Time Scheduling

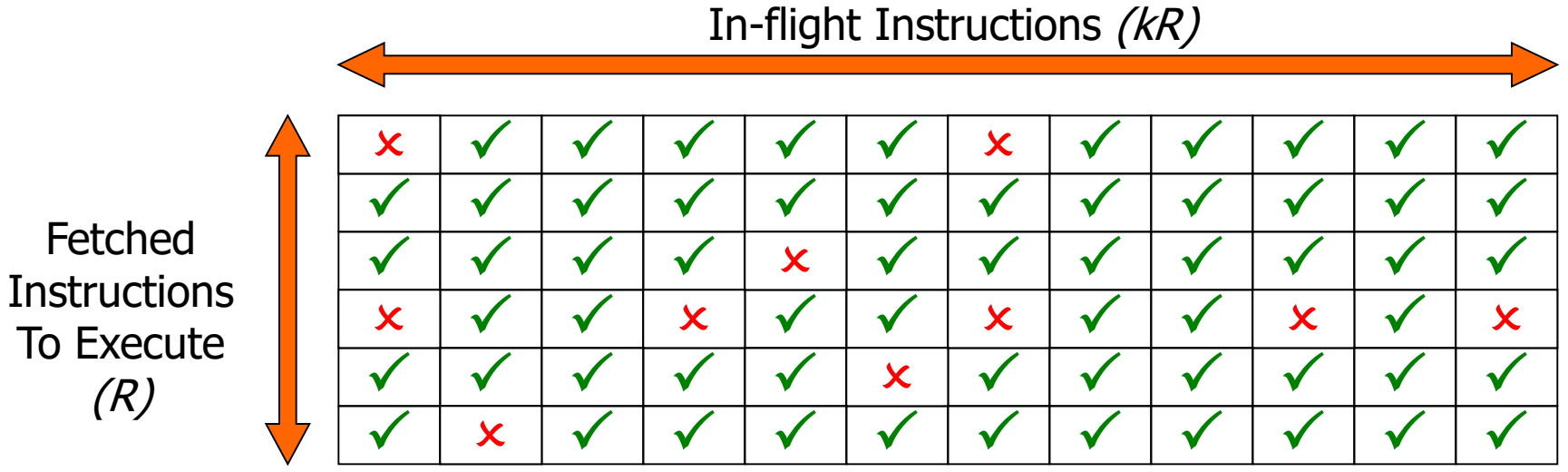
- What does it mean to schedule?
  - It means to decide **WHEN** and **WHERE** each instruction is executed
- Scheduling happens at run time in superscalars and it happens **exclusively** at compile time in VLIWs
- Run time scheduling in superscalars requires considerable resources in the processor hardware
  - Reservation stations and reorder buffer
  - Renaming registers and various sorts of mapping tables
  - Etc.

Dynamic Scheduler

- Large amount of logic, significant area cost
  - PowerPC 750 Instruction Sequencer is approx. 70% of the area all execution units! (Integer units + Load/ Store units + FP unit)
- Cycle time limited by scheduling logic
- Design verification extremely complex
- Design-for-Testability (DFT) complex
  - Very complex irregular logic

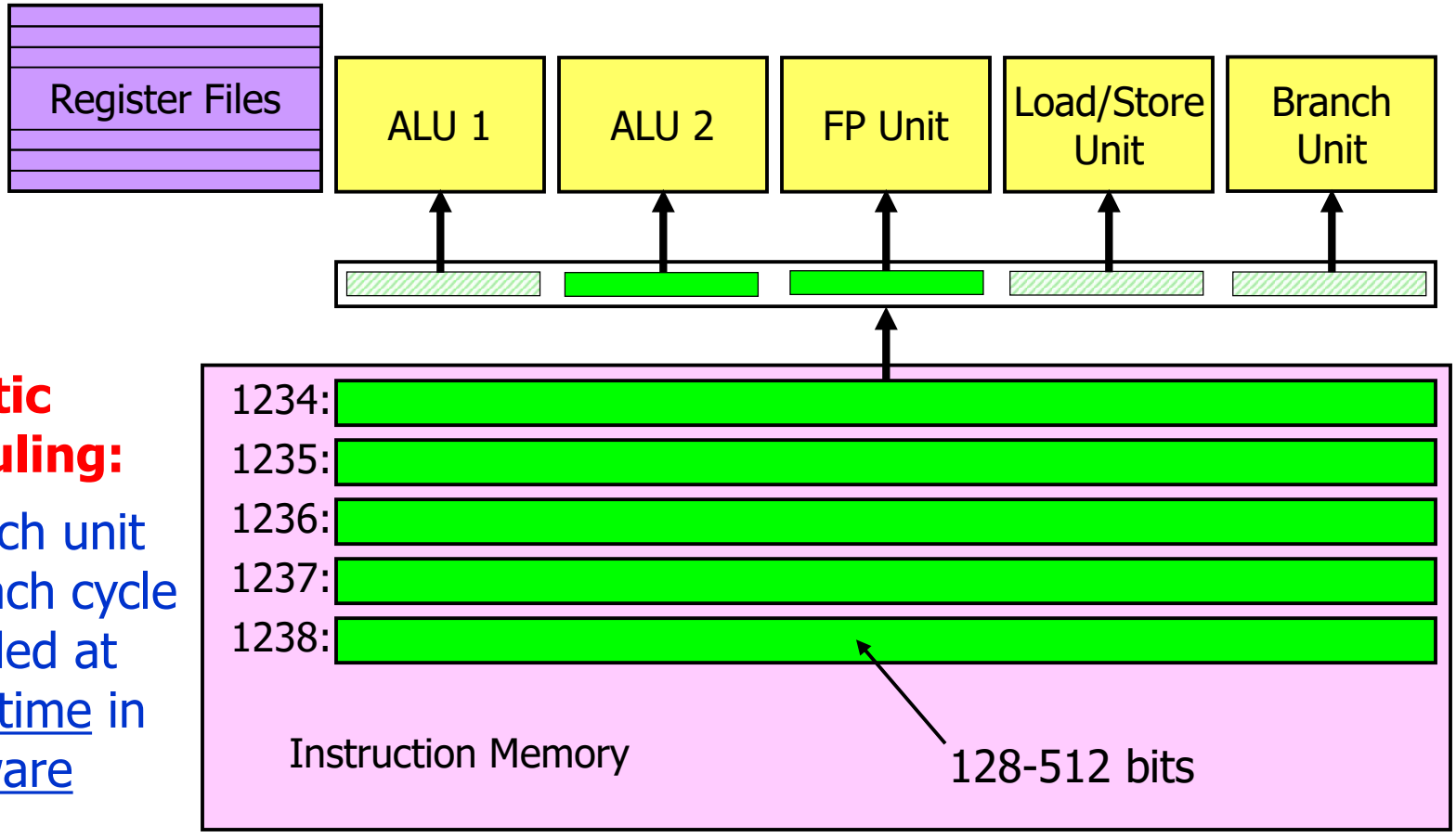
Dynamic Scheduler

- **Scheduling complexity** (e.g., checking dependences) is typically of the order of the square in the issue rate *(R)*



➔ Strong limit to ILP exploitation

(Statically Scheduled) Very Long Instruction Word Processor



Static Scheduling:

What each unit does in each cycle is decided at compile time in software

How to exploit Instruction Level Parallelism

- **Superscalar Processor**
  - **Hardware** detects parallelism among instructions
  - Scheduling is first performed at compile time, but with very loose information on the architecture the program will be run on
  - Final scheduling is performed at **run time**

- **VLIW (or EPIC) Processor**
  - **Software** detects parallelism among instructions
  - Scheduling is performed at **compile time**

Traditional Code vs. VLIW Code

Traditional

1000:	op 1
1001:	op 2
1002:	op 3
1003:	op 4
1004:	op 5
1005:	op 6

cycles != instructions

latency-independent semantics

(Unit-Assumed Latency)

VLIW

1000:	op 1	op 6	op 7	NOP
1001:	NOP	NOP	op 3	op 4
1002:	NOP	op 2	NOP	NOP
1003:	NOP	op 5	op 12	NOP
1004:	NOP	NOP	NOP	op 17
1005:	NOP	NOP	op 8	op 16

cycles = instructions

latency-dependent semantics

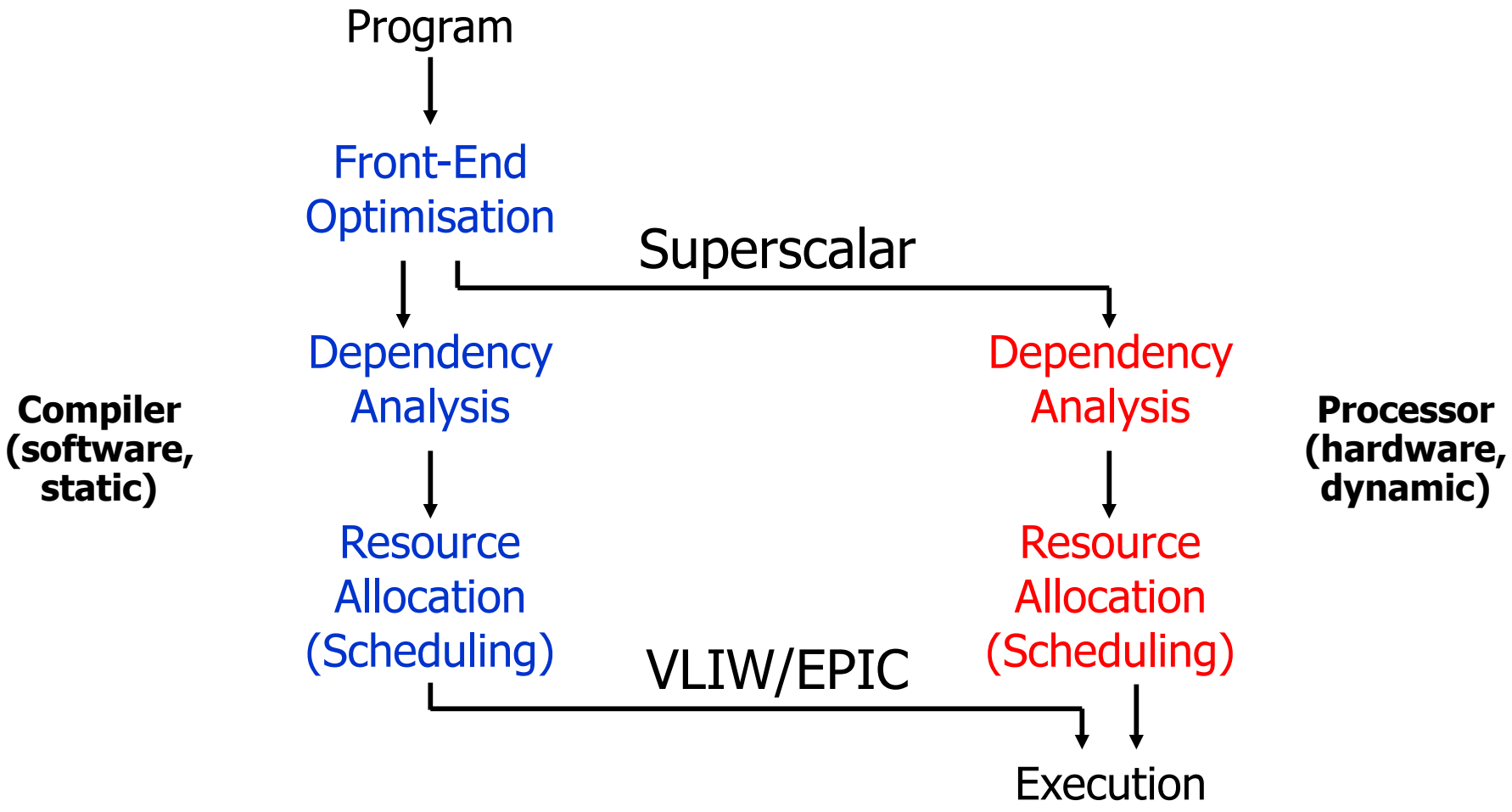
(Non Unit-Assumed Latency)

VLIW Main Advantage: Low Hardware Complexity

- **Area Advantage**: No need for the hardware used in superscalars for dynamic dependence analysis → **more execution units**

- **Timing Advantage**: No need for complex dependence analysis every cycle → **clock frequency can be higher**

A Different Split between Software and Hardware



A Different Split between Software and Hardware

```
…
Loop: ld     $f0, ($r1)
      addd   $f4, $f0, $f2
      sd     ($r1), $f4
      subi   $r1, $r1, 8
      bnez   $r1, Loop
…
```

```
…
Loop: ld     $f0, ($r1)
      addd   $f4, $f0, $f2
      sd     ($r1), $f4
      ld     $f6, ($r1-8)
      addd   $f8, $f6, $f2
      sd     ($r1-8), $f8
      ld     $f10, ($r1-16)
      addd   $f12, $f10, $f2
      sd     ($r1-16), $f12
      ld     $f14, ($r1-24)
      addd   $f16, $f14, $f2
      sd     ($r1-24), $f16
      ld     $f18, ($r1-32)
      addd   $f20, $f18, $f2
      sd     ($r1-32), $f20
      subi   $r1, $r1, 40
      bnez   $r1, Loop
…
```

```
…
Loop: ld     $f0, ($r1)
      ld     $f6, ($r1)
      ld     $f10, ($r1)        addd   $f4, $f0, $f2
      ld     $f14, ($r1)        addd   $f8, $f6, $f2
      ld     $f18, ($r1)        addd   $f12, $f10, $f2
      sd     ($r1), $f4         addd   $f16, $f14, $f2
      sd     ($r1-8), $f8       addd   $f20, $f18, $f2
      sd     ($r1-16), $f12
      sd     ($r1-24), $f16
      sd     ($r1-32), $f20
      subi   $r1, $r1, 40
      bnez   $r1, Loop
…
```

SW

(= Complier)

Source code available
(higher level information)
Global analysis possible
(variable lifecycle, etc.)

VLIW

HW

(= Instruction Scheduler)

Run-time information available
(actual data, addresses, pointers, etc.)

Challenges of VLIW

- Compiler Technology
  - Compiler now responsible for scheduling
  - Most severe limitation until recently (VLIW idea is around since the 70s!)
- Binary Incompatibility
  - Consequence of the larger exposure of the microarchitecture (= implementation choices) in the architecture (e.g., NUAL semantics)
- Code Bloating
  - All those NOPs occupy memory space and thus cost
  - But there are also other reasons!...

2

The Code Bloating Problem

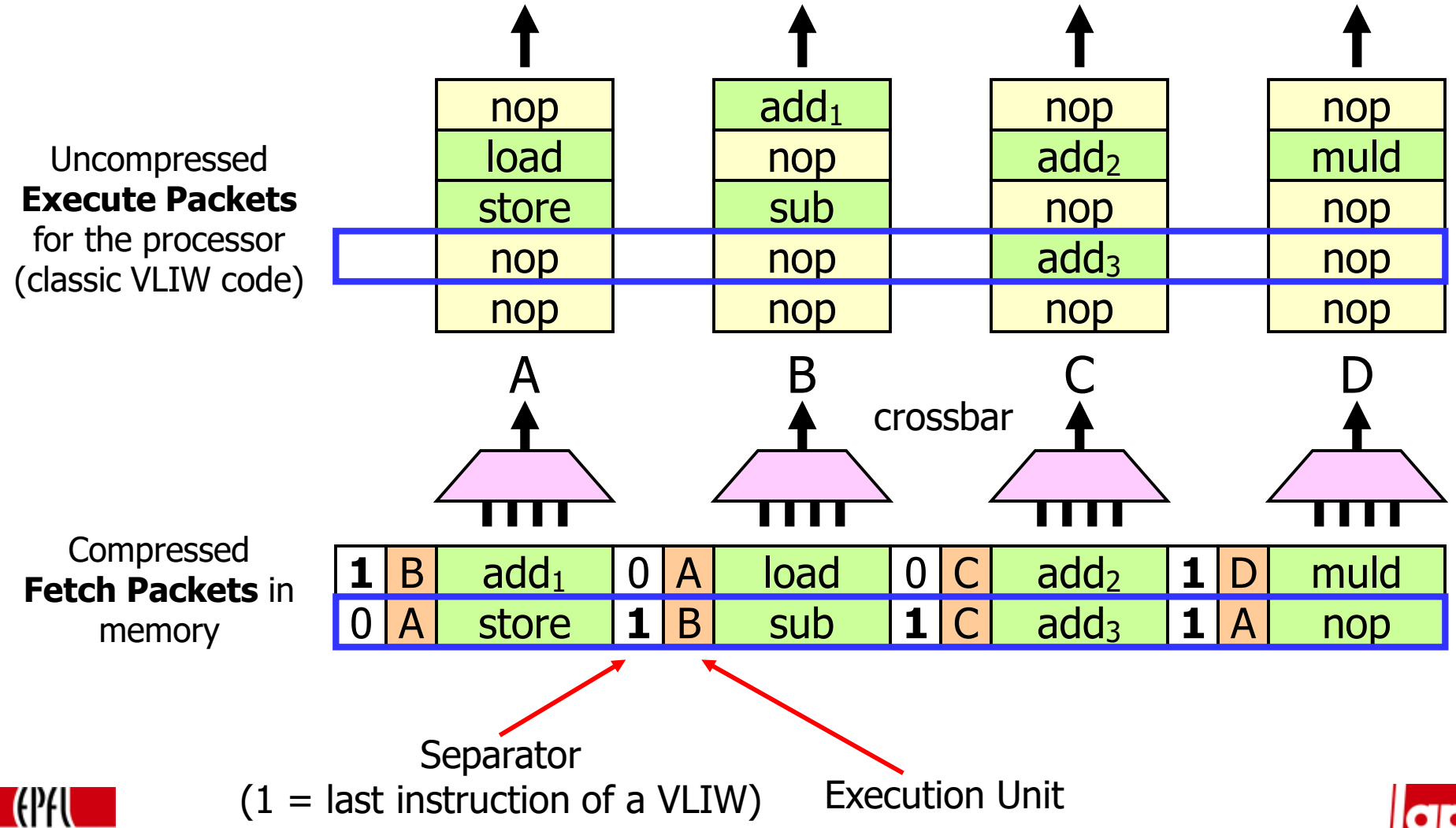
(Memory Is Not That Cheap—and More...)

Larger Code Is a Serious Problem

1000:	op 1	op 6	op 7	NOP
1001:	NOP	NOP	op 3	op 4
1002:	NOP	op 2	NOP	NOP
1003:	NOP	op 5	op 12	NOP
1004:	NOP	NOP	NOP	op 17
1005:	NOP	NOP	op 8	op 16

- In a first approximation, the problem is due to the explicit NOPs
- Not just a DRAM cost issue (main memory is cheap…), but has weird impacts on cache performance (size, cache pollution, associativity, etc.)

Code Compression: Differentiate Fetch Packet and Execute Packet



Typical VLIW Code Compression

- Instructions are encoded in a less straightforward way
  - Separator bit = 0: next operation is in parallel
  - Separator bit = 1: next operation is sequential
  - Unit number: specifies where to execute operation
- Price to pay for shorter code:
  - Fetch/Decode logic more complex
  - Crossbar for shipping operations to the right FU, complexity proportional to n^2

Hardware was supposed to be trivial and O(n)...

Code Bloating Solved?

- A trivial but significant reason for bloating is removed
- More fundamental and difficult to overcome reasons exist which still increase significantly the code size
- See later…

3

The Binary Compatibility Problem

(Not everybody likes—or can—recompile…)

NUAL Semantics Assumes More...

1000:	op 1	op 6	op 7	NOP
1001:	NOP	NOP	op 3	op 4
1002:	NOP	op 2	NOP	NOP
1003:	NOP	op 5	op 12	NOP
1004:	NOP	NOP	NOP	op 17
1005:	NOP	NOP	op 8	op 16

cycles = instructions

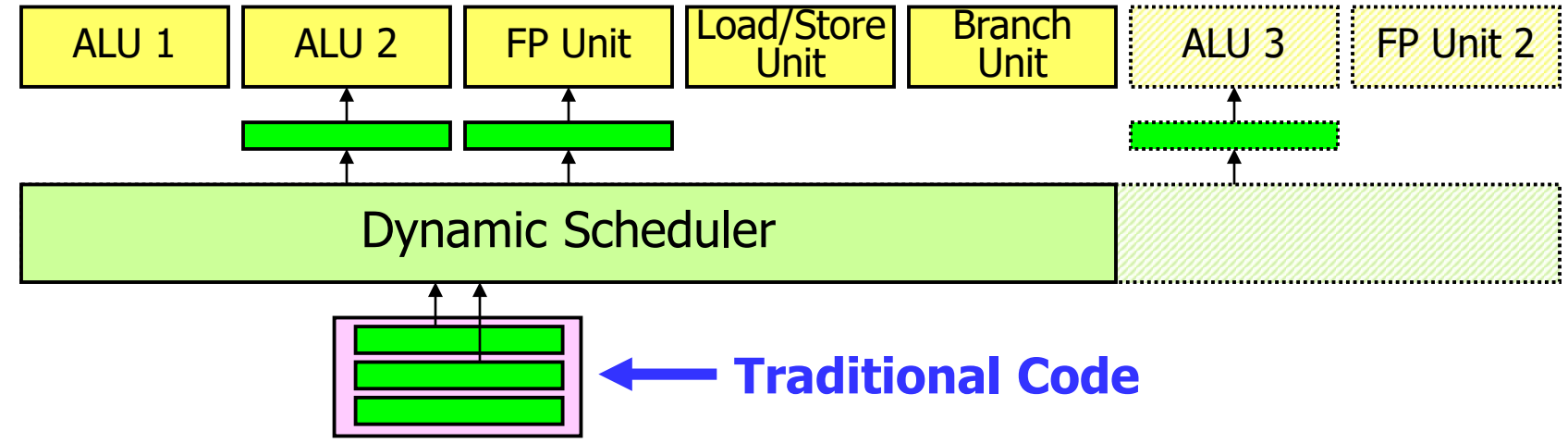
latency-dependent semantics

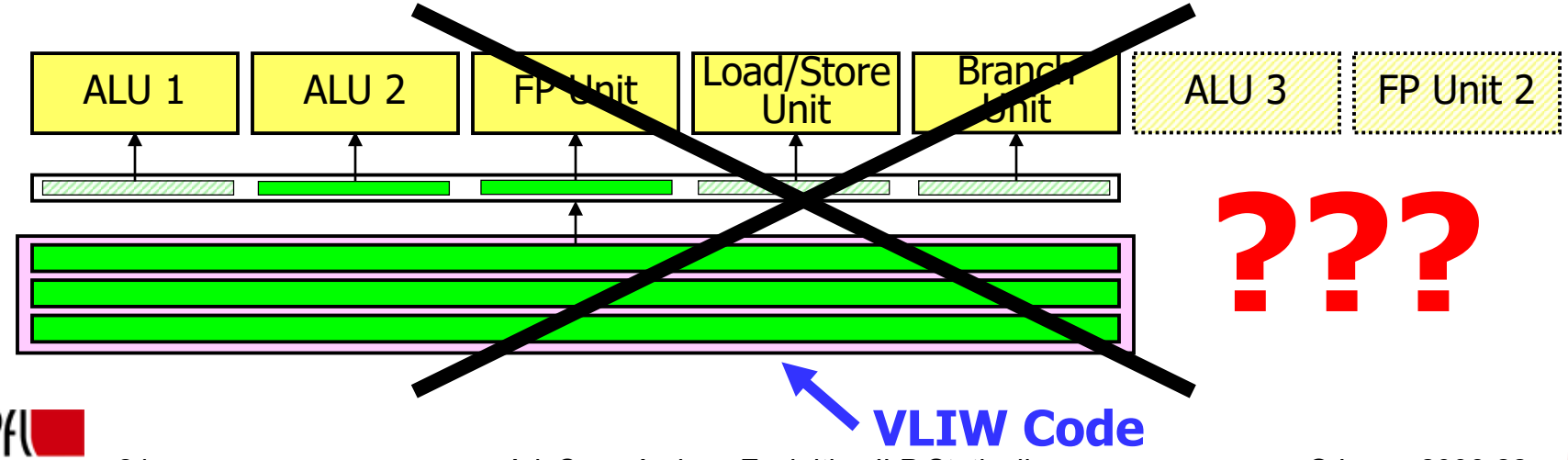
(Non Unit-Assumed Latency)

More information is now implicit in the code:

1. **Instruction latencies**—used to enforce correct handling of data dependencies
2. **Available hardware parallelism**—units scheduled on each cycle

VLIW Binary Is Incompatible with More Aggressive Implementations



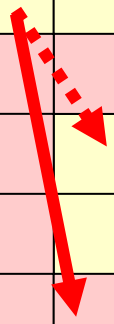


VLIW Binary Incompatibility

- More subtle sources of incompatibility
  - Changes in instruction latencies—e.g., load latencies increases *(logic-memory gap)*
- No fully satisfactory solution exists today
- Partial or research solutions:
  - Recompile (possible in some kind of systems—not for consumer PC market…)
  - Special VLIW coding/restrictions
  - Dynamic Binary Translation is emerging—see future course

Problem #1
Latency Cannot Increase

1000:	op 1	op 6	op 7	NOP
1001:	NOP	NOP	op 3	op 4
1002:	NOP	op 2	NOP	NOP
1003:	NOP	op 5	op 12	NOP
1004:	NOP	NOP	NOP	op 17
1005:	NOP	NOP	op 8	op 16



cycles = instructions

latency-dependent semantics

(Non Unit-Assumed Latency)

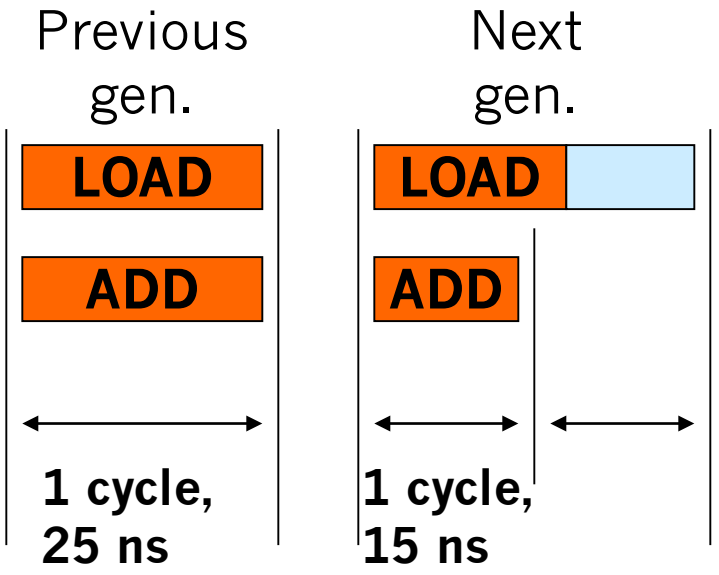
Trivially, higher latency may **violate data dependencies**

- E.g., the operands of "op 2" are no longer available if the latency of "op 1" increases.

Why Latency Could Ever Increase?

```
0   load   R2,(R5)
1   add    R3,R3,R2
```

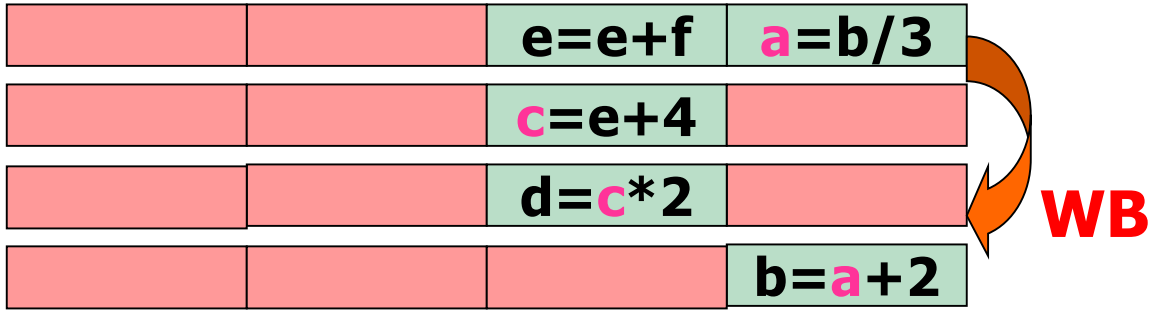
load: 1 cycle → 2 cycles



- Latency can sometimes increase in next generation machines:
  - E.g. memory / logic growing gap

Problem #2
Latency Cannot Decrease Either!

```
a = b / 3;
b = a + 2;
e = e + f;
c = e + 4;
d = c * 2;
```



If division takes 3 cycles, and addition takes 1 cycle…

Values **c** and **a** can be assigned to the same physical register in this schedule (**a** is dead while **c** is alive)

If, in the next generation, division takes only 2 cycles ➔ **wrong result!**

4

The Compiler Problem

(Not Just a New Compiler...)

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Typical Code May Have Limited ILP

- Example:

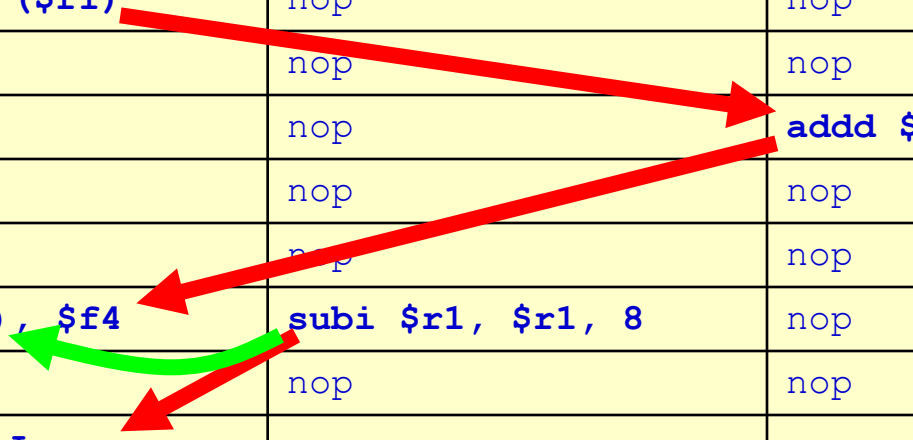
```
Loop: ld     $f0, ($r1)        // read array elem.
      addd   $f4, $f0, $f2     // add constant
      sd     ($r1), $f4        // write array elem.

      subi   $r1, $r1, 8       // next element
      bnez   $r1, Loop
```

- Schedule on a VLIW processor
  - Slot 1: Load/Store Unit or Branch Unit
  - Slot 2: ALU
  - Slot 3: Floating-Point Unit
- Latencies:
  - Load/Store → 2 cycles
  - Integer → 2 cycles
  - Branch → 2 cycles
  - Floating Point → 3 cycles

Typical Code May Have Limited ILP

- Scheduled VLIW code:

Load/Store/Branch Unit	ALU	Floating-Point Unit	
`ld $f0, ($r1)`	nop	nop	Cycle 1
nop	nop	nop	Cycle 2
nop	nop	`addd $f4, $f0, $f2`	Cycle 3
nop	nop	nop	Cycle 4
nop	nop	nop	Cycle 5
`sd ($r1), $f4`	`subi $r1, $r1, 8`	nop	Cycle 6
nop	nop	nop	Cycle 7
`bnez $r1, Loop`	nop	nop	Cycle 8
nop	nop	nop	Cycle 9

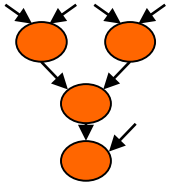


- Execution time for $r1 = 80:
  - 80 / 8 = 10 iterations; 9 cycles per iteration → **90 cycles**

Fighting Dependencies

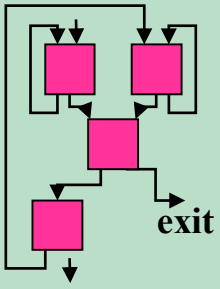
- Parallel execution is limited by the need to find **independent instructions**
- We need to deal with both **data and control** dependencies
- Data:

```
a=b+c;
d=a+d;
```



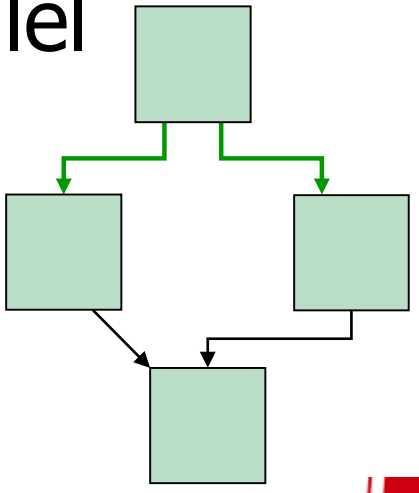
- Control:

```
if (a==b)
    d=c+d;
```



Overcoming Control Dependencies: Predicated Execution

- If
  - We have abundant resources (machine parallelism), and
  - We do not care about power dissipation, etc. but just look for performance
- We can execute all paths in parallel without making a choice



Predicated Execution

- Remove branches via **If Conversion**:

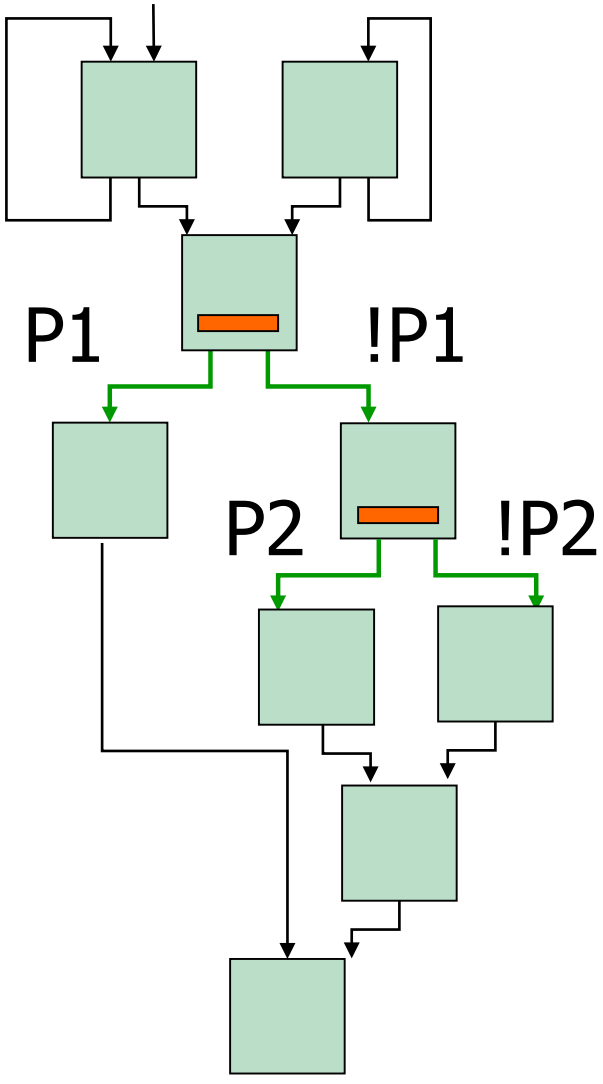
```
if (a==b)    c=2*d   else   c=3*d
```

becomes

```
P1 = (a==b)
(P1) c=2*d          (!P1) c=3*d
```

- Introduce predicate P1 (outcome of jump)
- Instructions can now be all executed in parallel, but they are committed only if the relative predicate is true

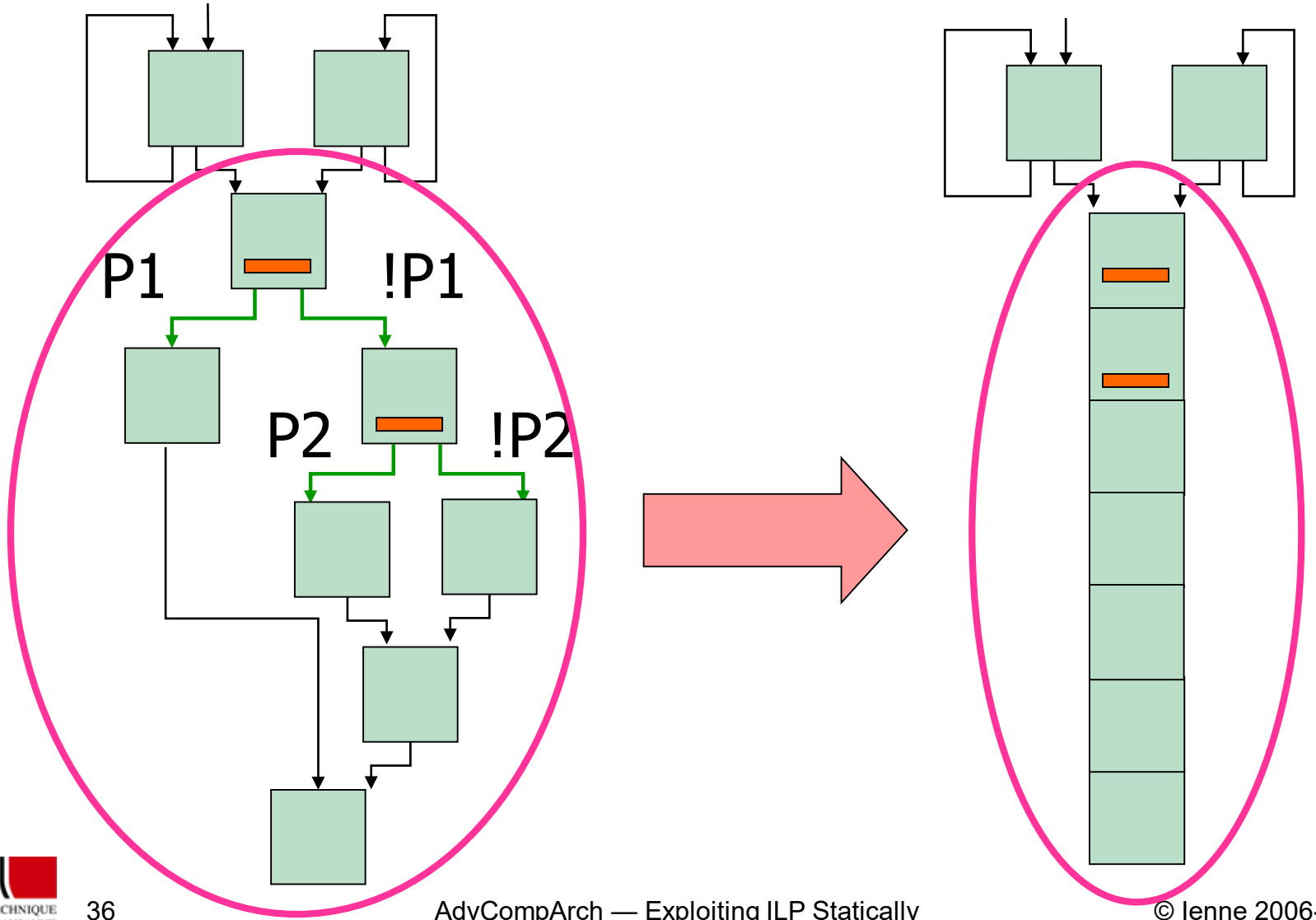
Predicated Execution Needs Architectural Support



We need:

1. An instruction (▬) to set the predicate
2. Predicate registers
3. An additional field in the instruction word
4. A way to check and delay exceptions

Predicated Execution Makes Basic Blocks Larger



Predication Support Can Be Partial or Full

- **Full**: all instructions can be executed conditionally
  - ARM (on the flags)
  - IA-64/Itanium (on predicate registers)
- **Partial**: typically a single conditional instruction
  - STMicroelectronics ST2xx: Select instruction
  - Alpha: Conditional Move instruction

Predication without Architectural Support...

- Before...

```
/* an excerpt from g72x.c */
/* g721encoder, mediabench */

anmant = (anmag == 0) ? 32:
   (anexp >= 0) ? anmag >> anexp: anmag << -anexp;
```

- After...

```
/* an excerpt from predicated g72x.c */
/* g721encoder, mediabench */

p2 = -(anmag == 0); p3 = -(anexp >= 0);
anmant = (32 & p2) | ((anmag >> anexp) & ~p2 & p3) |
         ((anmag << -anexp) & ~p2 & ~p3);
```

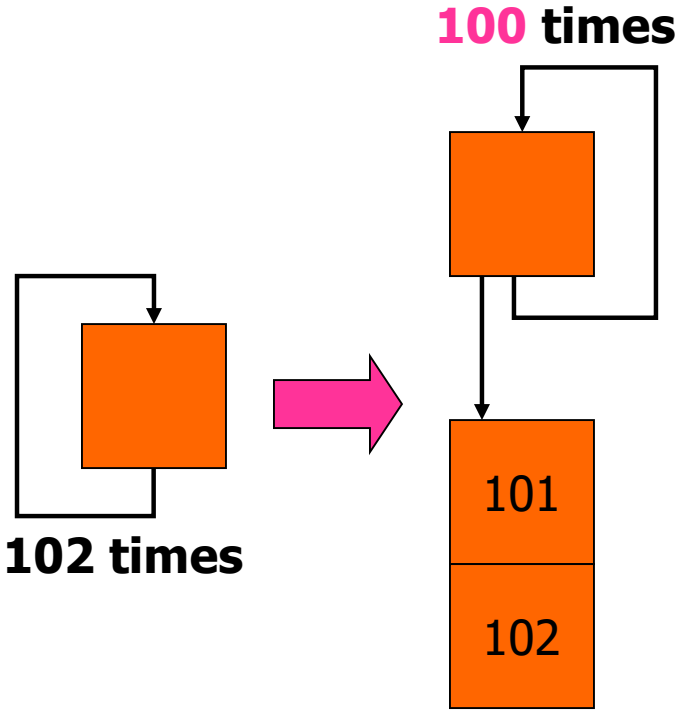
Predication without Architectural Support…

- Suppose that
  - Branches are very poorly predictable (p = 0.5)
  - Branches costs 1 or 5 cycles (taken/untaken)
  - Tests and other ALU ops cost 1 cycle
  - There are several ALUs available (e.g., 3)
- Trace of **normal** program is
  - Test → Branch → (Move || Test → Branch → Shift)
  - On average 1 + (1+5)/2 + ½ + (1 + (1+5)/2 + 1)/2 = **~7 cycles**
- Trace of **modified** program is
  - 2 Tests → 2 Negs → 2 Nots → 2 Shifts, 5 Ands, 3 Ors
  - Ideally some 16/3 = **~5-6 cycles**
- Predication **could** in **special cases** be also a programming *trick* for normal processors not supporting it in hardware!…

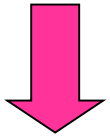
Overcoming Control Dependencies: Loop Transformations

- Loops are often the most important part of code (in terms of fraction of total time)
- Loops bodies can be transformed so that more parallelism can be exploited
  - Loop peeling
  - Loop fusion
  - Loop distribution
  - Loop unrolling
  - Software pipelining, etc.

Loop Peeling



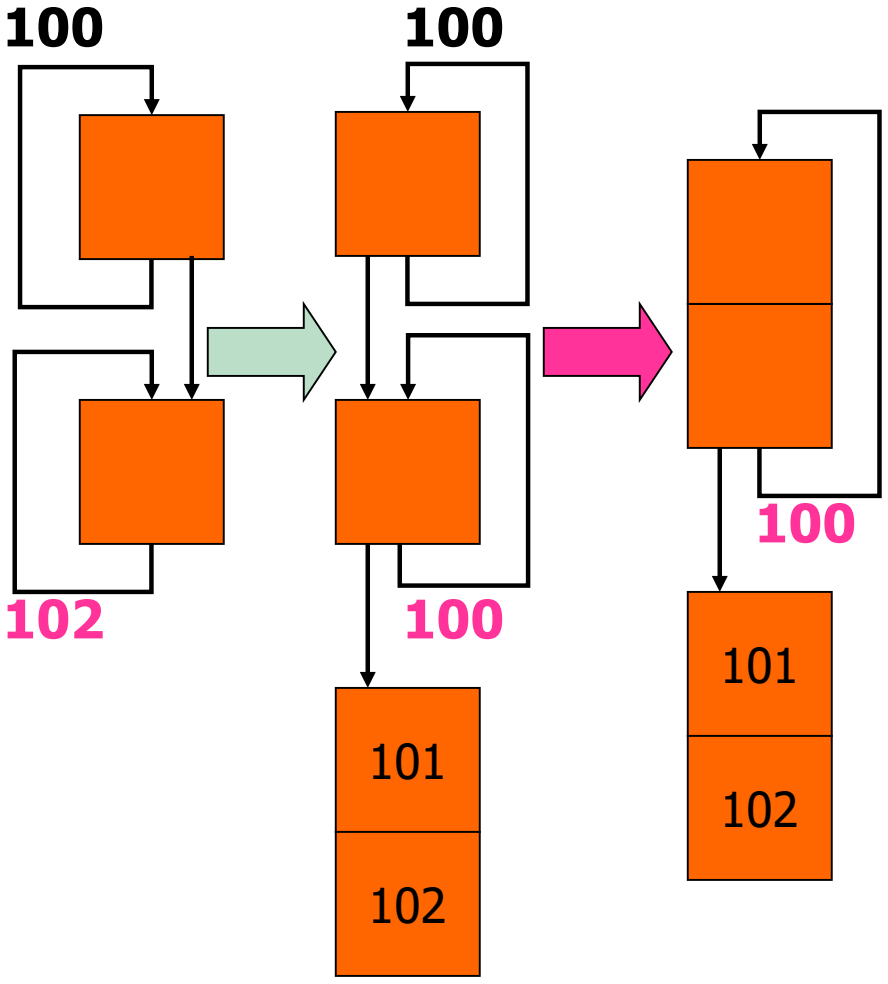

```
for (i=0;i<102;i++)
        a[i]=a[i-1]+c;
```



```
for (i=0;i<100;i++)
    a[i]=a[i-1]+c;
a[100]=a[99]+c;
a[101]=a[100]+c;
```

Used with fusion (next slide) to increase ILP

Loop Fusion

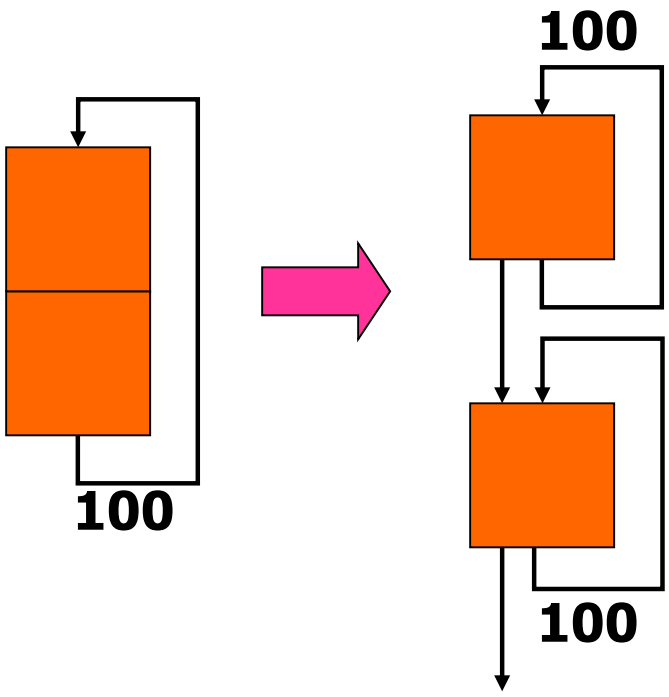



```
for (i=0;i<102;i++)
        b[i]=b[i-2]+c;
for (j=0;j<100;j++)
        a[j]=a[j]*2;
```

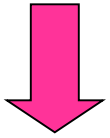
```
for (i=0;i<100;i++) {
    b[i]=b[i-2]+c;
    a[i]=a[i]*2;}
a[100]=a[100]*2;
a[101]=a[101]*2;
```

Now a and b can be computed in parallel

Loop Distribution



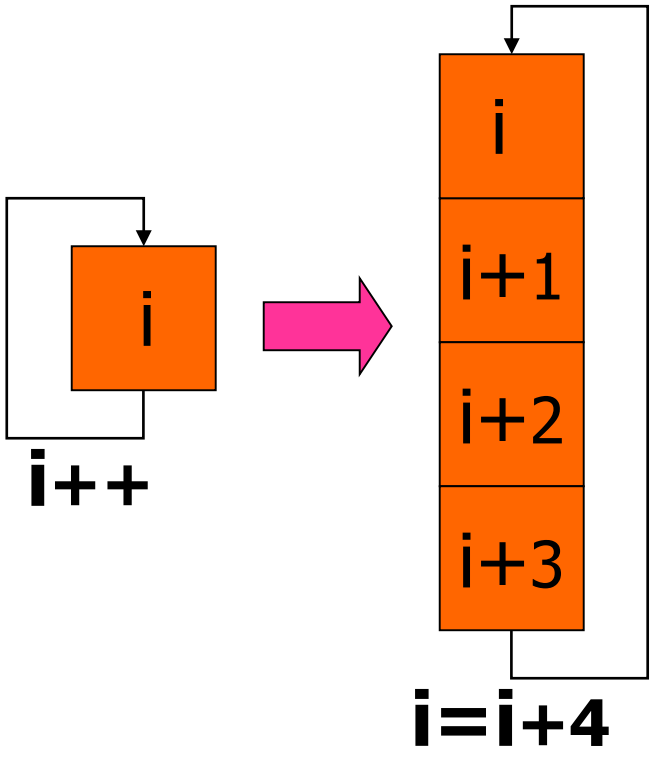

```
for (i=0;i<100;i++) {
        b[i]=b[i-1]+c;
        a[i]=b[i]+2;}
```



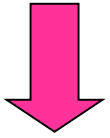
```
for (i=0;i<100;i++)
        b[i]=b[i-1]+c;
for (i=0;i<100;i++)
        a[i]=b[i]+2;
```

Now the second loop can be unrolled and parallelised

Creating Larger Loop Bodies: Loop Unrolling




```
for (i=0;i<100;i++)
        a[i]=a[i]+c;
```



```
for (i=0;i<100;i=i+4) {
    a[i]=a[i]+c;
    a[i+1]=a[i+1]+c;
    a[i+2]=a[i+2]+c;
    a[i+3]=a[i+3]+c;}
```

4 times less jumps and more scope for ILP (larger basic block)

Example of Loop Unrolling

- Example:

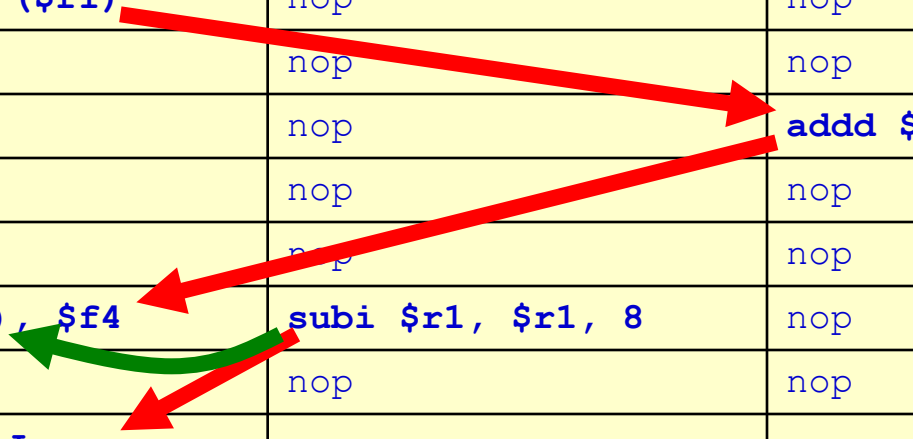
```
Loop: ld     $f0, ($r1)        // read array elem.
      addd   $f4, $f0, $f2     // add constant
      sd     ($r1), $f4        // write array elem.

      subi   $r1, $r1, 8       // next element
      bnez   $r1, Loop
```

- Schedule on a VLIW processor
  - Slot 1: Load/Store Unit or Branch Unit
  - Slot 2: ALU
  - Slot 3: Floating-Point Unit
- Latencies:
  - Load/Store → 2 cycles
  - Integer → 2 cycles
  - Branch → 2 cycles
  - Floating Point → 3 cycles

Before Unrolling

- Scheduled VLIW code:

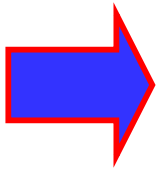
Load/Store/Branch Unit	ALU	Floating-Point Unit	
`ld $f0, ($r1)`	nop	nop	Cycle 1
nop	nop	nop	Cycle 2
nop	nop	`addd $f4, $f0, $f2`	Cycle 3
nop	nop	nop	Cycle 4
nop	nop	nop	Cycle 5
`sd ($r1), $f4`	`subi $r1, $r1, 8`	nop	Cycle 6
nop	nop	nop	Cycle 7
`bnez $r1, Loop`	nop	nop	Cycle 8
nop	nop	nop	Cycle 9



- Execution time for $r1 = 80:
  - 80 / 8 = 10 iterations; 9 cycles per iteration → **90 cycles**

Loop Unrolling Idea

```
Loop: ld     $f0, ($r1)
      addd   $f4, $f0, $f2
      sd     ($r1), $f4

      subi   $r1, $r1, 8
      bnez   $r1, Loop
```



```
Loop: ld     $f0, ($r1)
      addd   $f4, $f0, $f2
      sd     ($r1), $f4

      ld     $f6, -8($r1)
      addd   $f8, $f6, $f2
      sd     -8($r1), $f8

      ld     $f10, -16($r1)
      addd   $f12, $f10, $f2
      sd     -16($r1), $f12

      ld     $f14, -24($r1)
      addd   $f16, $f14, $f2
      sd     -24($r1), $f16

      ld     $f18, -32($r1)
      addd   $f20, $f18, $f2
      sd     -32($r1), $f20

      subi   $r1, $r1, 40
      bnez   $r1, Loop
```

- Replicate body
- Update references
- Rename registers
- etc.

Unrolled and Rescheduled

Load/Store/Branch Unit	ALU	Floating-Point Unit	
`ld $f0, ($r1)`	nop	nop	Cycle 1
`ld $f6, -8($r1)`	nop	nop	Cycle 2
`ld $f10, -16($r1)`	nop	`addd $f4, $f0, $f2`	Cycle 3
`ld $f14, -24($r1)`	nop	`addd $f8, $f6, $f2`	Cycle 4
`ld $f18, -32($r1)`	nop	`addd $f12, $f10, $f2`	Cycle 5
`sd ($r1), $f4`	nop	`addd $f16, $f14, $f2`	Cycle 6
`sd -8($r1), $f8`	nop	`addd $f20, $f18, $f2`	Cycle 7
`sd -16($r1), $f12`	nop	nop	Cycle 8
`sd -24($r1), $f16`	nop	nop	Cycle 9
`sd -32($r1), $f20`	`subi $r1, $r1, 40`	nop	Cycle 10
nop	nop	nop	Cycle 11
`bnez $r1, Loop`	nop	nop	Cycle 12
nop	nop	nop	Cycle 13

- Now 80 / (5*8) = 2 iterations; 13 cycles per iteration → **26 cycles** (vs. 90 cycles, more than 3x faster!)

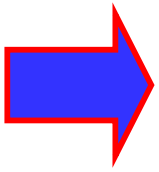
No Architectural Extension Required (So Far…)

- Some techniques seen here require architectural extensions
  - Predication
  - Branch prediction
  - …
- Others do not
  - Basic loop transformations (peeling, fusion,…)
  - Loop unrolling
  - …
- Yet, they may have an indirect impact on architectural needs—e.g., more registers

VLIW Code Bloating Revisited...

- VLIW code fundamentally larger than standard code: not only NOPs are explicit, but aggressive unrolling multiplies real instructions
- Compare last example: **39** words vs. **5**! more than **50% are NOPs**!

```
ld   $f0, ($r1)
addd $f4, $f0, $f2
sd   ($r1), $f4
subi $r1, $r1, 8
bnez $r1, Loop
```



ld $f0, ($r1)	nop	nop
ld $f6, -8($r1)	nop	nop
ld $f10, -16($r1)	nop	addd $f4, $f0, $f2
ld $f14, -24($r1)	nop	addd $f8, $f6, $f2
ld $f18, -32($r1)	nop	addd $f12, $f10, $f2
sd ($r1), $f4	nop	addd $f16, $f14, $f2
sd -8($r1), $f8	nop	addd $f20, $f18, $f2
sd -16($r1), $f12	nop	nop
sd -24($r1), $f16	nop	nop
sd -32($r1), $f20	subi $r1, $r1, 40	nop
nop	nop	nop
bnez $r1, Loop	nop	nop
nop	nop	nop

Beyond Loop Unrolling: Software Pipelining

- Restructure the body of the loop so that more parallelism can be extracted
- Put different tasks from different iterations in the same iteration (to exploit ILP)

BUT: Do not increase code size

(as loop unrolling does)

Software Pipelining

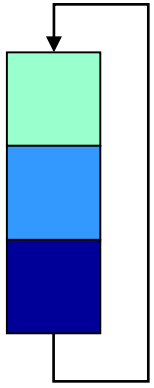
- Consider the following simple C code snippet:

```
for (i=0, i<7, i++) {
    c[i] = a[i]+1;
}
```

load a[i]

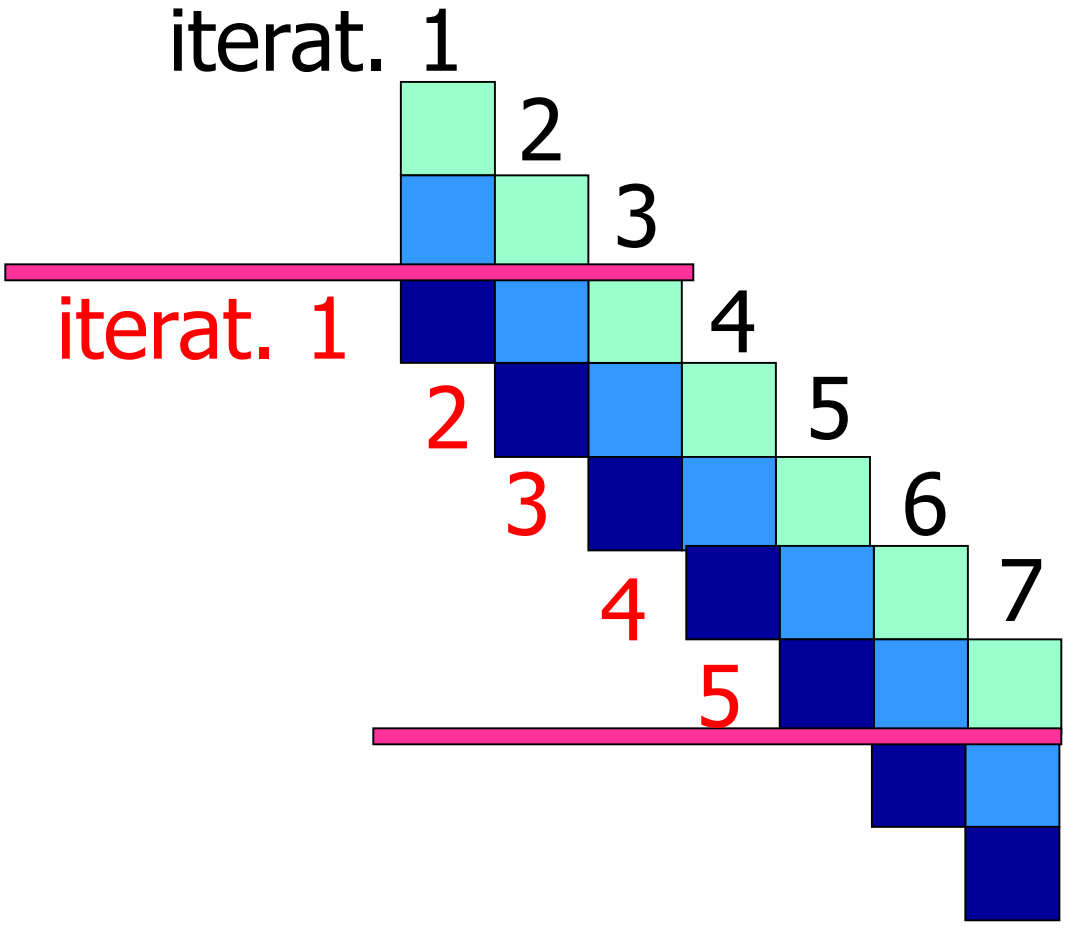
add a[i], #1

store c[i]

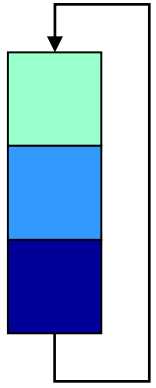


- The three corresponding instructions are dependent, they cannot be executed in parallel
- Goal: **restructure the loop**, so that some ILP can be exploited

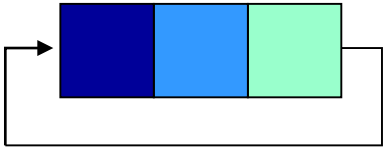
Software Pipelining Idea



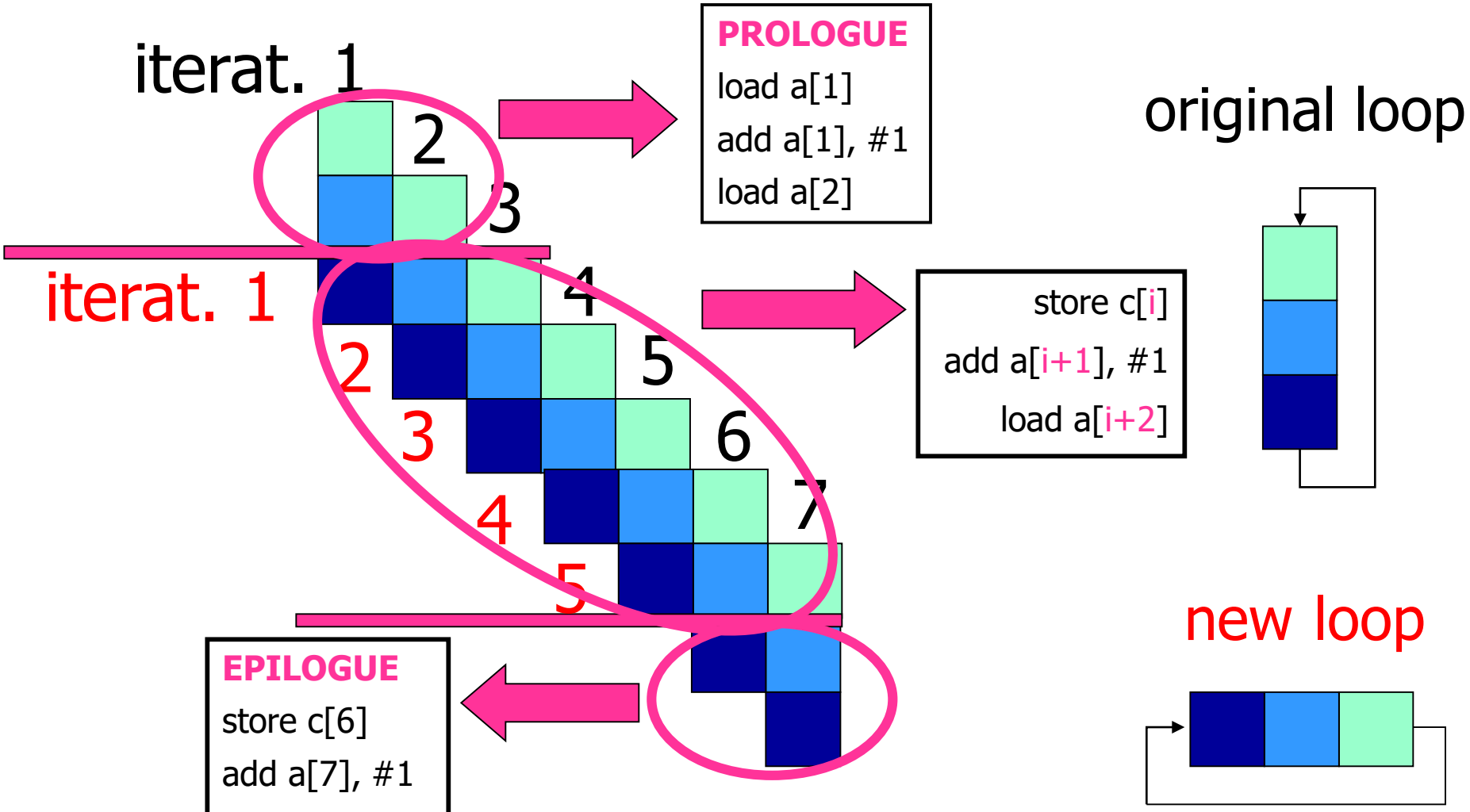
original loop



new loop



Software Pipelining
Prologue, Body, and Epilogue



SW Pipelining Example

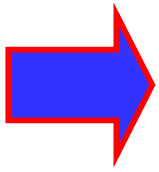
- Same example:

```
Loop: ld     $f0, ($r1)        // read array elem.
      addd   $f4, $f0, $f2     // add constant
      sd     ($r1), $f4        // write array elem.

      subi   $r1, $r1, 8       // next element
      bnez   $r1, Loop
```

- Schedule on a VLIW processor
  - Slot 1 **and 2**: Load/Store Unit or Branch Unit
  - Slot 3: ALU
  - Slot 4: Floating-Point Unit
- Latencies:
  - Load/Store → 2 cycles
  - Integer → 2 cycles
  - Branch → 2 cycles
  - Floating Point → 3 cycles

Again, Unroll the Loop

```
Loop: ld     $f0, ($r1)
      addd   $f4, $f0, $f2
      sd     ($r1), $f4

      subi   $r1, $r1, 8
      bnez   $r1, Loop
```



```
Loop: ld     $f0, ($r1)
      addd   $f4, $f0, $f2
      sd     ($r1), $f4

      ld     $f6, -8($r1)
      addd   $f8, $f6, $f2
      sd     -8($r1), $f8

      ld     $f10, -16($r1)
      addd   $f12, $f10, $f2
      sd     -16($r1), $f12

      ld     $f14, -24($r1)
      addd   $f16, $f14, $f2
      sd     -24($r1), $f16

      ld     $f18, -32($r1)
      addd   $f20, $f18, $f2
      sd     -32($r1), $f20

      subi   $r1, $r1, 40
      bnez   $r1, Loop
```

- Replicate body
- Update references
- Rename registers
- etc.

Unrolled Loop Schedule

LOAD Unit	STORE Unit	ALU	Floating-Point Unit	
LD #0				Cycle 1
LD #1				Cycle 2
LD #2			ADDD #0	Cycle 3
LD #3			ADDD #1	Cycle 4
LD #4			ADDD #2	Cycle 5
LD #5	SD #0		ADDD #3	Cycle 6
LD #6	SD #1		ADDD #4	Cycle 7
LD #7	SD #2		ADDD #5	Cycle 8
	SD #3		ADDD #6	Cycle 9
	SD #4		ADDD #7	Cycle 10
	SD #5			Cycle 11
	SD #6			Cycle 12
	SD #7			Cycle 13

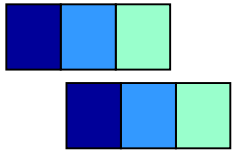
Identify the Regular Kernel

LOAD /Branch Unit	STORE Unit	ALU	Floating-Point Unit
`ld $f0, ($r1)`			
`ld $f6, -8($r1)`			
`ld $f0, -16($r1)`			`addd $f4,$f0,$f2`
`ld $f6, -24($r1)`			`addd $f8,$f6,$f2`
`ld $f0, -32($r1)`			`addd $f12,$f0,$f2`
`ld $f6, -40($r1)`	`sd 0($r1), $f4`		`addd $f4,$f10,$f2`
`ld $f10, -48($r1)`	`sd -8($r1), $f8`		`addd $f8,$f14,$f2`
`ld $f14, -56($r1)`	`sd -16($r1), $f12`	`subi $r1,$r1,24`	`addd $f12,$f6,$f2`
`bnez $r1, Loop`			
	`sd 0($r1), $f4`		`addd $f4,$f10,$f2`
	`sd -8($r1), $f8`		`addd $f8,$f14,$f2`
	`sd -16($r1), $f12`		
	`sd -24($r1), $f4`		
	`sd -32($r1), $f8`		

Modified SW Pipelining Example (All Unit Latencies)

LOAD /Branch Unit (latency ONE)	STORE Unit (latency ONE)	ALU (latency ONE)	Floating-Point Unit (latency ONE)
`ld $f0, ($r1)`			
`ld $f0, -8($r1)`			`addd $f4,$f0,$f2`
`ld $f0, -16($r1)`	`sd 0($r1), $f4`	`subi $r1,$r1,8`	`addd $f4,$f0,$f2`
`bnez $r1, Loop`			
	`sd 0($r1), $f4`		`addd $f4,$f0,$f2`
	`sd -8($r1), $f4`		

Why “SW Pipelining”?



HW pipelining

- Fetch (i+2)
- Decode (i+1)
- Execute (i)

Instructions advancing in parallel

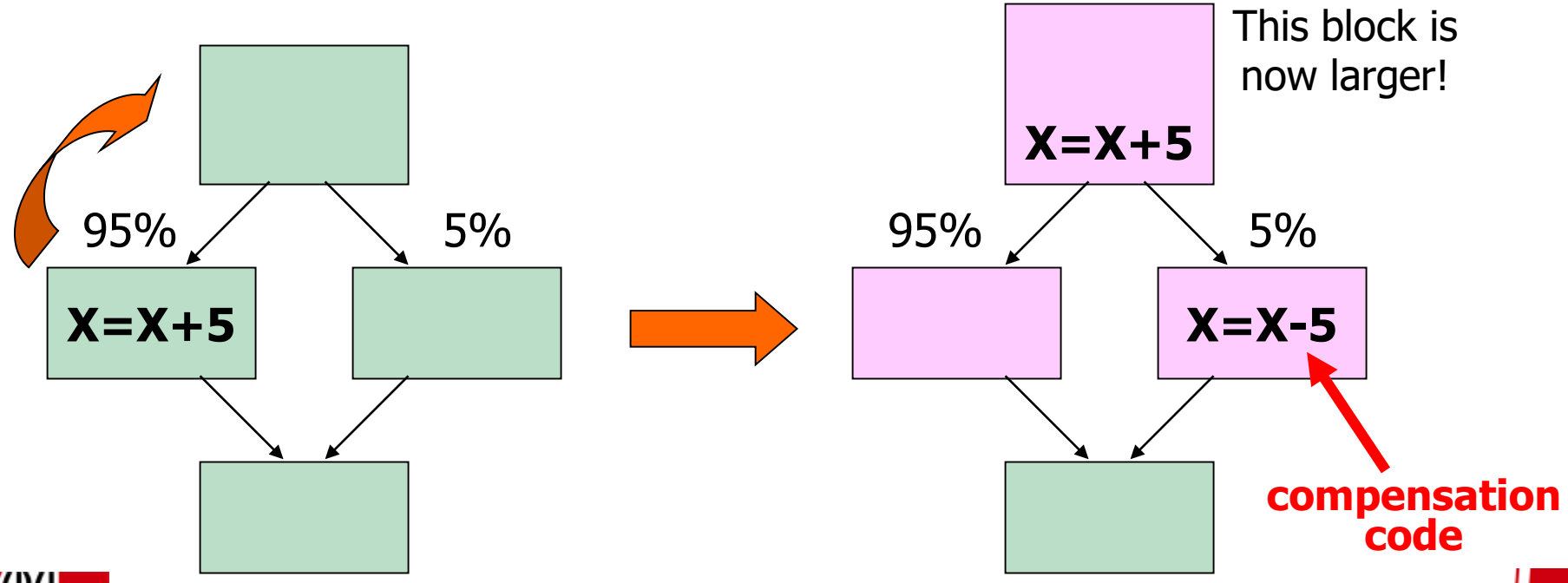
SW pipelining

- Load (i+2)
- Add (i+1)
- Store (i)

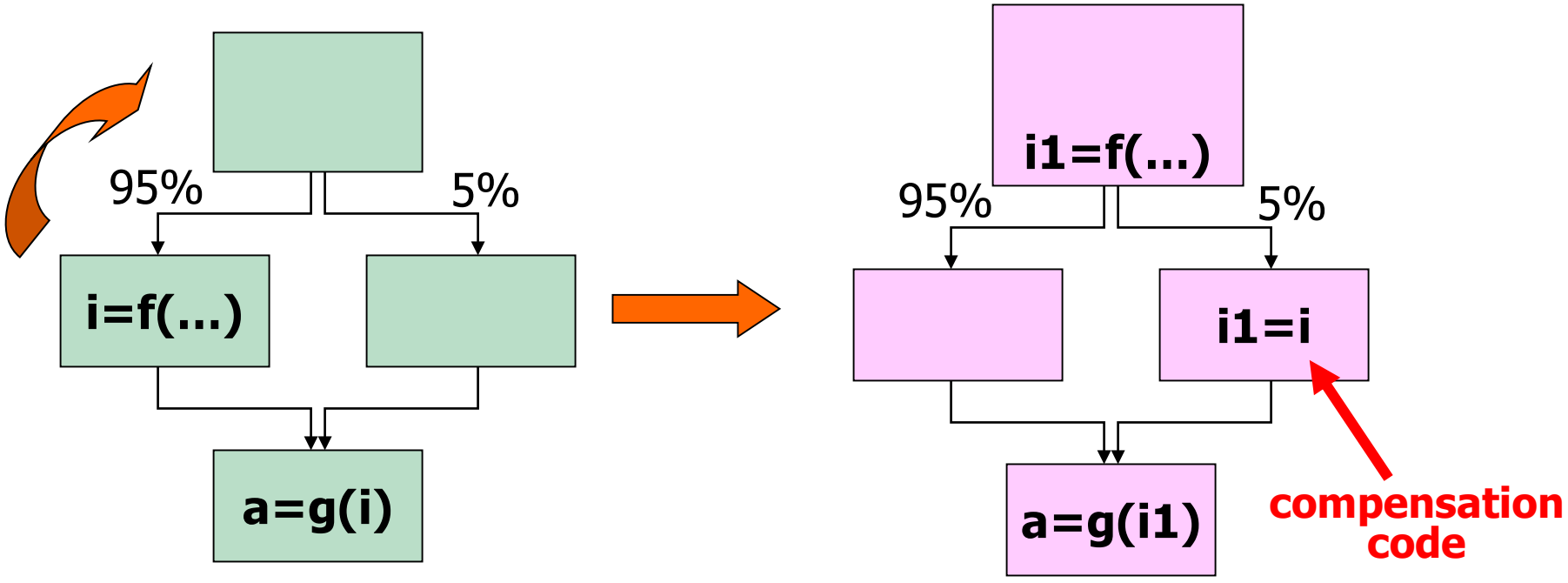
Iterations advancing in parallel

Overcoming Nonloop Control Dependences: Trace Scheduling

- Early technique: published by Fisher in 1981
- Optimise the **most probable path** by increasing the size of basic blocks (→ more chances to find ILP)
- Add compensation code in less probable paths
- Beyond basic blocks: region-based scheduling



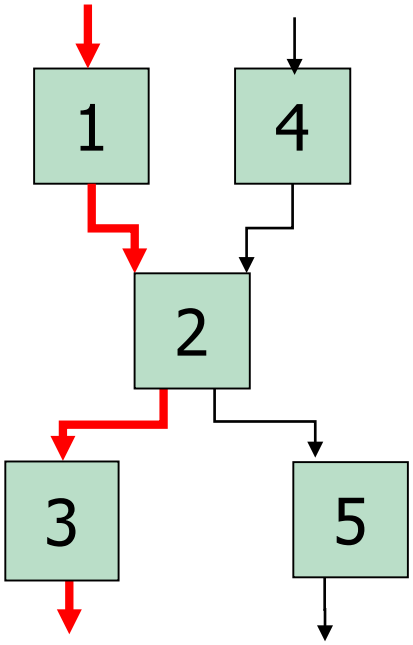
Register Renaming as a Way to Compensate



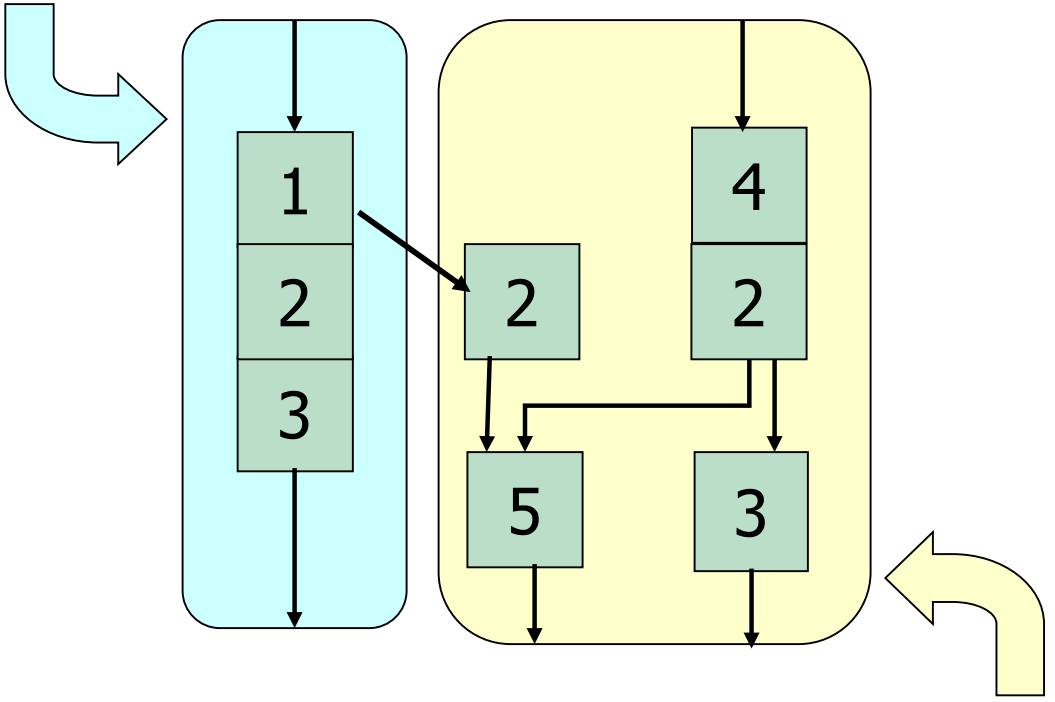
- Register renaming ensures that semantics is correct in every trace

  → But, again, we need more registers…

Trace Scheduling



Most probable trace
will be optimised



All possible (but less probable) traces are there, **plus** some **compensation code**

What Is Trace Scheduling? Static Speculation

- By moving instructions across branches to optimise probable path, we have done **speculation**
- **Dynamic (run time) speculation** is one of the most significant ingredients of superscalar performance
- Trace Scheduling is a form of **static (compile time) speculation**, and so are superblocks, hyperblocks,…

➔ **Region-Based Scheduling**

Run Time vs. Compile Time Speculation

- At **run time**: it is the hw that does it
- At **compile time**: the compiler schedules the speculated instruction before the branch ➔ It is speculated with respect to the original code, but in the resulting code one cannot really see it as being speculated
- It is what happens in **trace scheduling** and **superblock scheduling**

Compile Time Speculation

- **Register renaming** to ensure that correction code source operands are preserved
- Because of **exceptions**, you need to either:
  1. **Avoid Errors**: Speculate only instructions which cannot raise exceptions (but one wants to speculate loads!)
  2. **Resolve Errors**: Add a special field in the opcode (Poison bit,…) that says when an instruction has been speculated (see IA-64)

Architectural Needs for Run- vs. Compile-Time Speculation

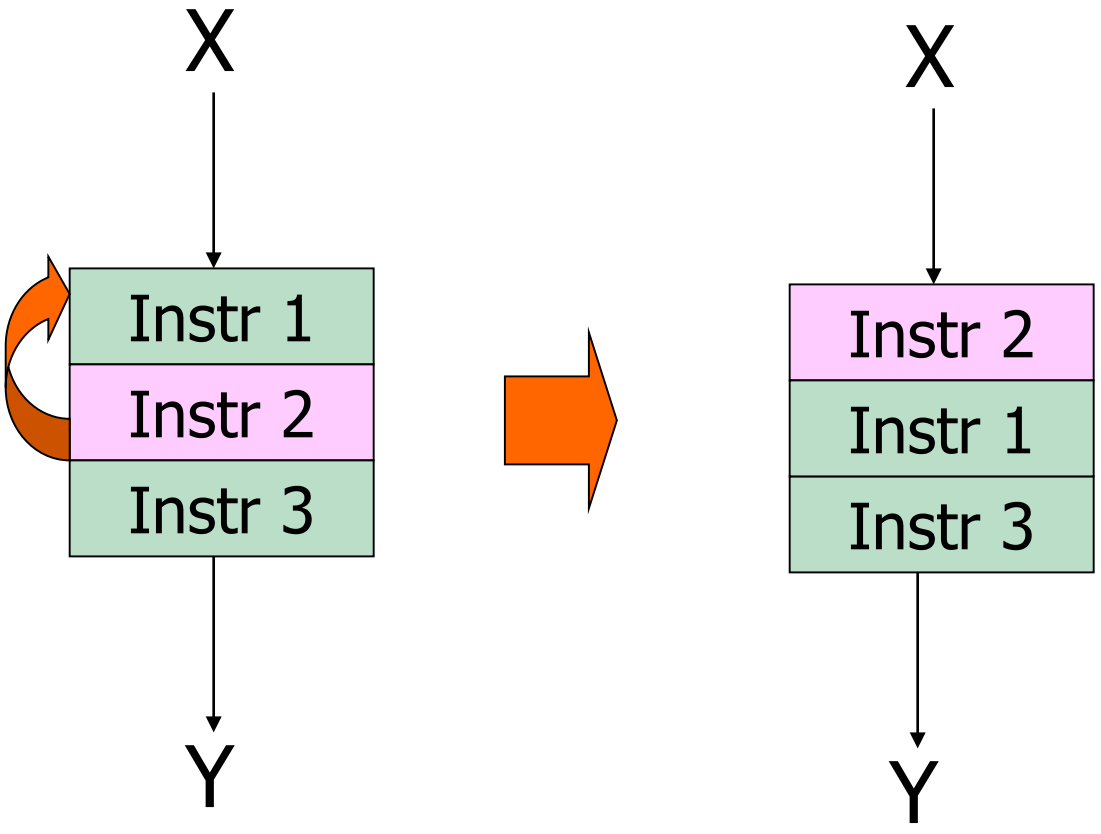
	Run-Time Speculation (superscalar)	Compile-Time Speculation (VLIW)
Where to speculate?	Predictors	Nothing! Profiling
How to nullify instructions?	Reorder buffer/ Commit unit	Nothing! Register renaming
How to handle exceptions?	Reorder buffer/ Commit unit	Poison bits/ speculative opcodes

Most pressure is on the compiler
But not everything can be done by compilers!

Compensation Code

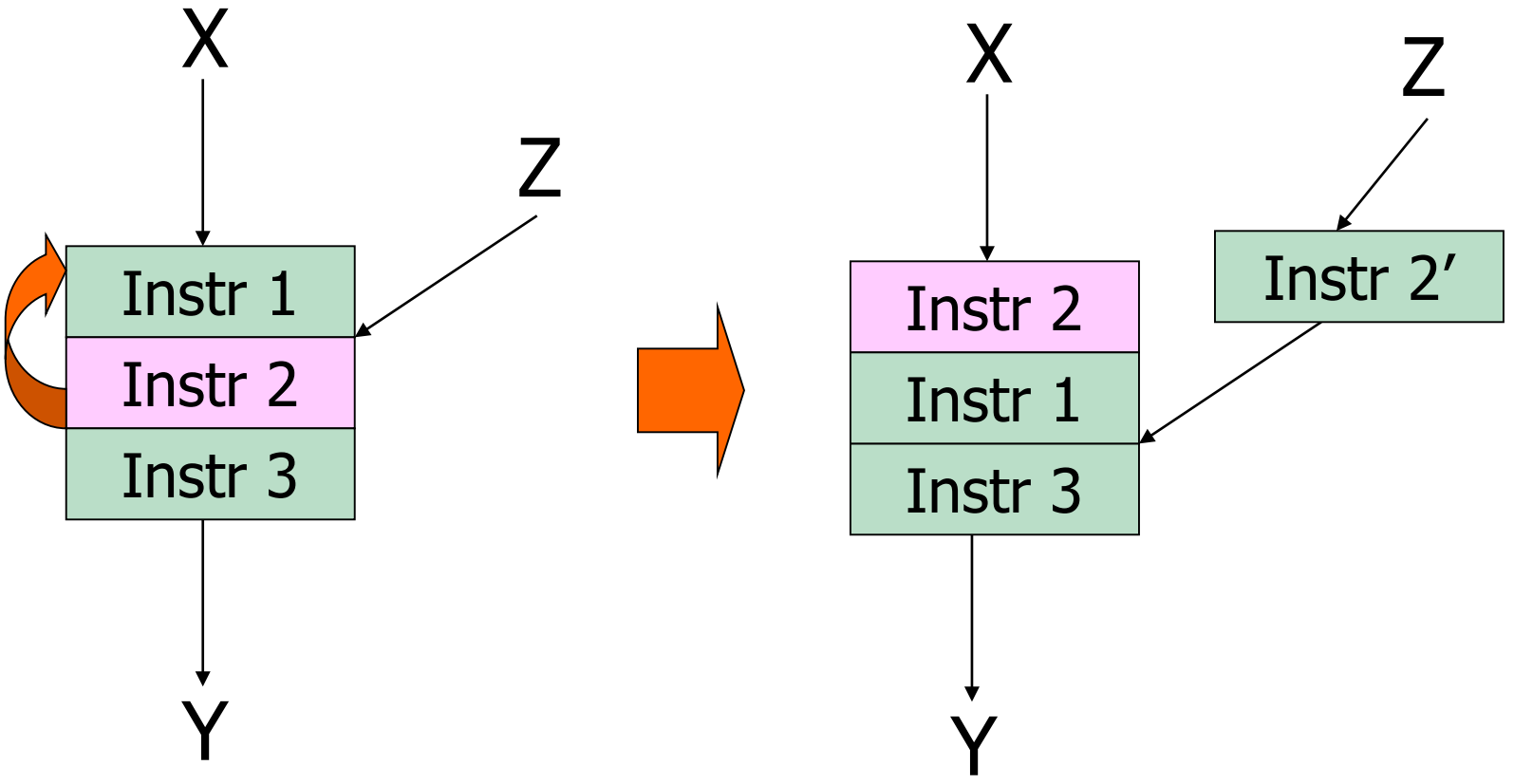
- Set of techniques to restore the correct flow of data and control because of global code motion
- 4 cases are possible:
  - No compensation (straight-line code)
  - Join compensation
  - Split compensation
  - Join/Split compensation

No Compensation



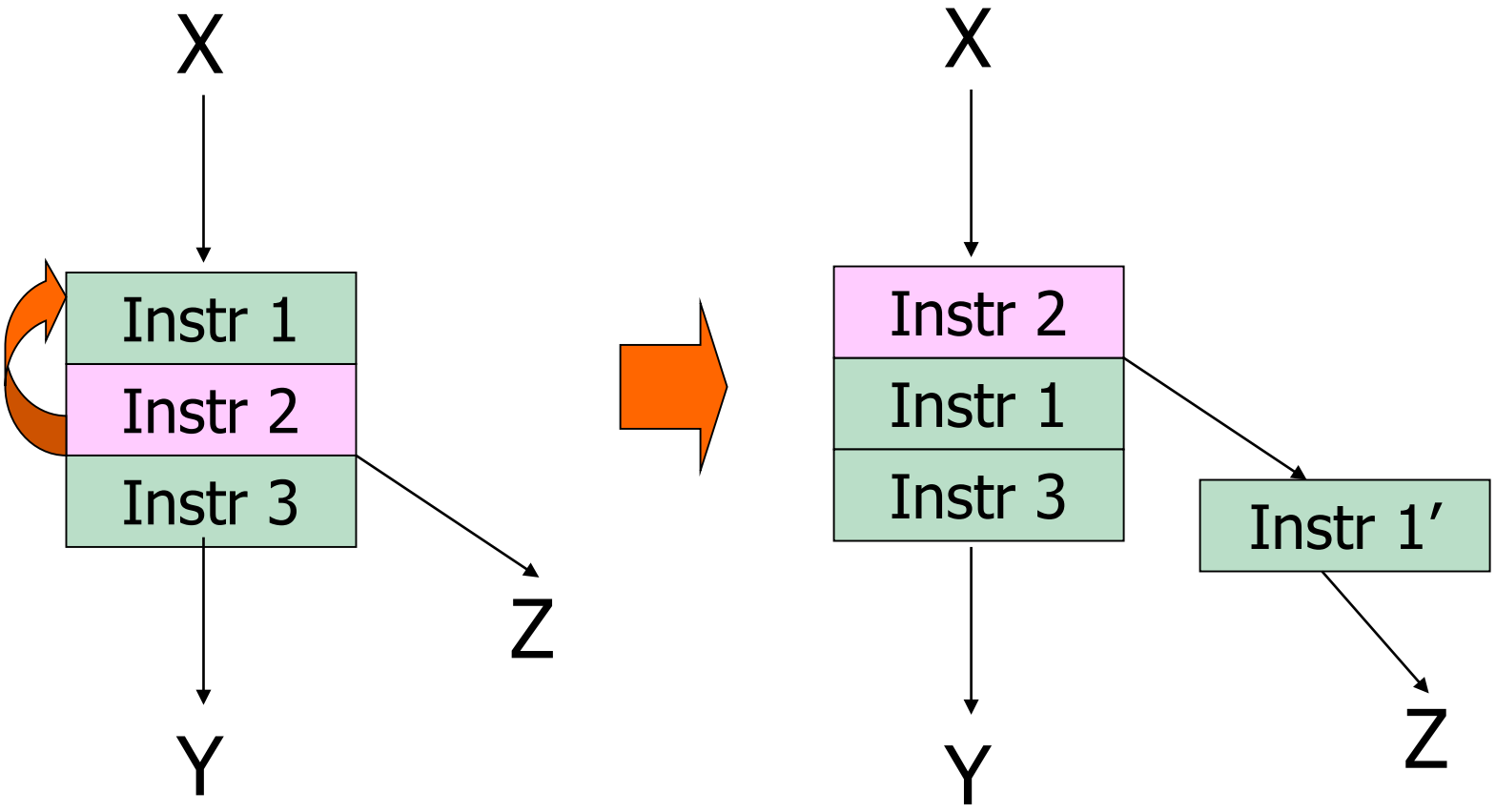
Swap 2 and 1, in a basic block

Join Compensation



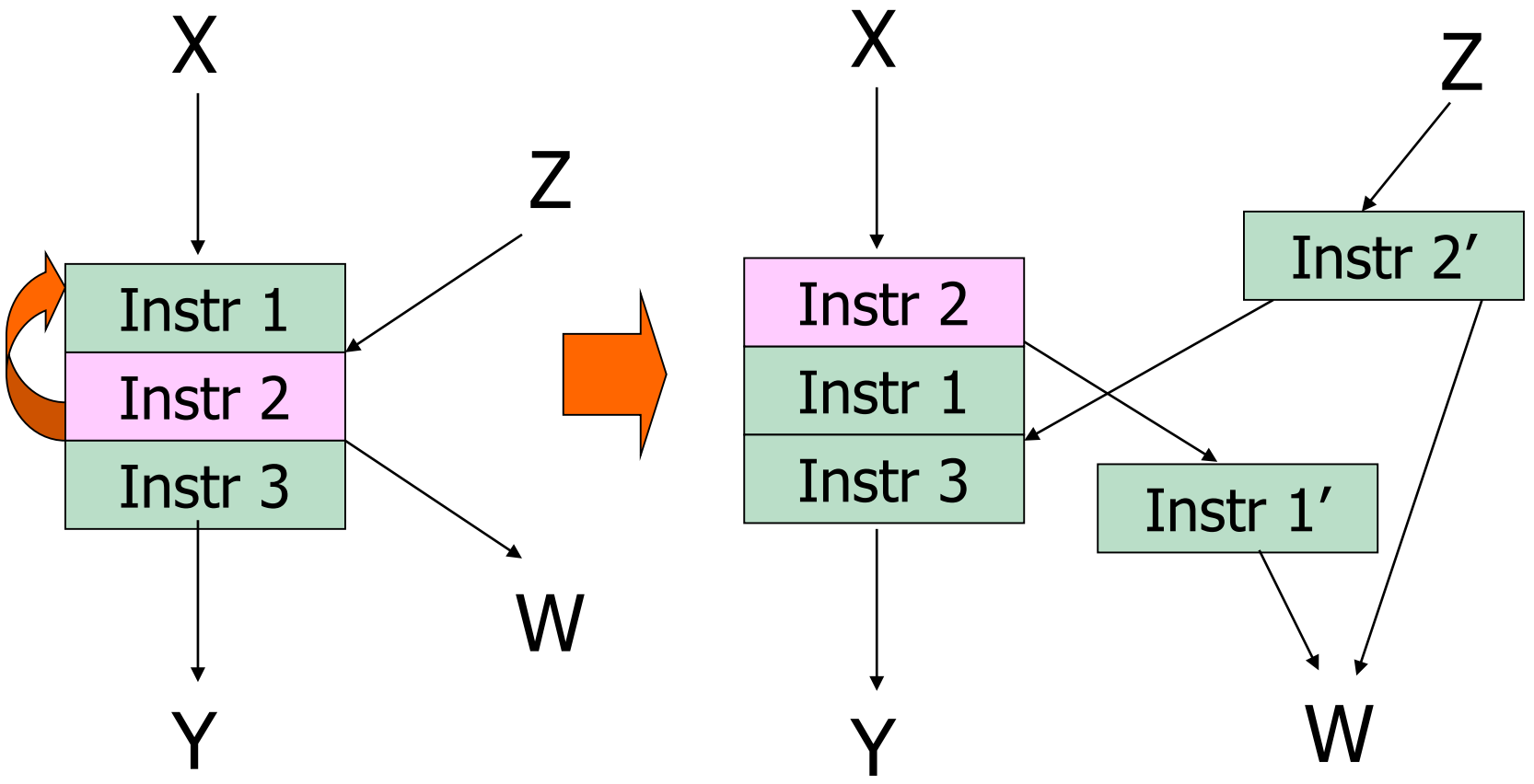
Swap 2 and 1, where 2 is a join

Split Compensation



Swap 2 and 1, where 2 is a split

Join/Split Compensation



Swap 2 and 1, where 2 is a join and a split

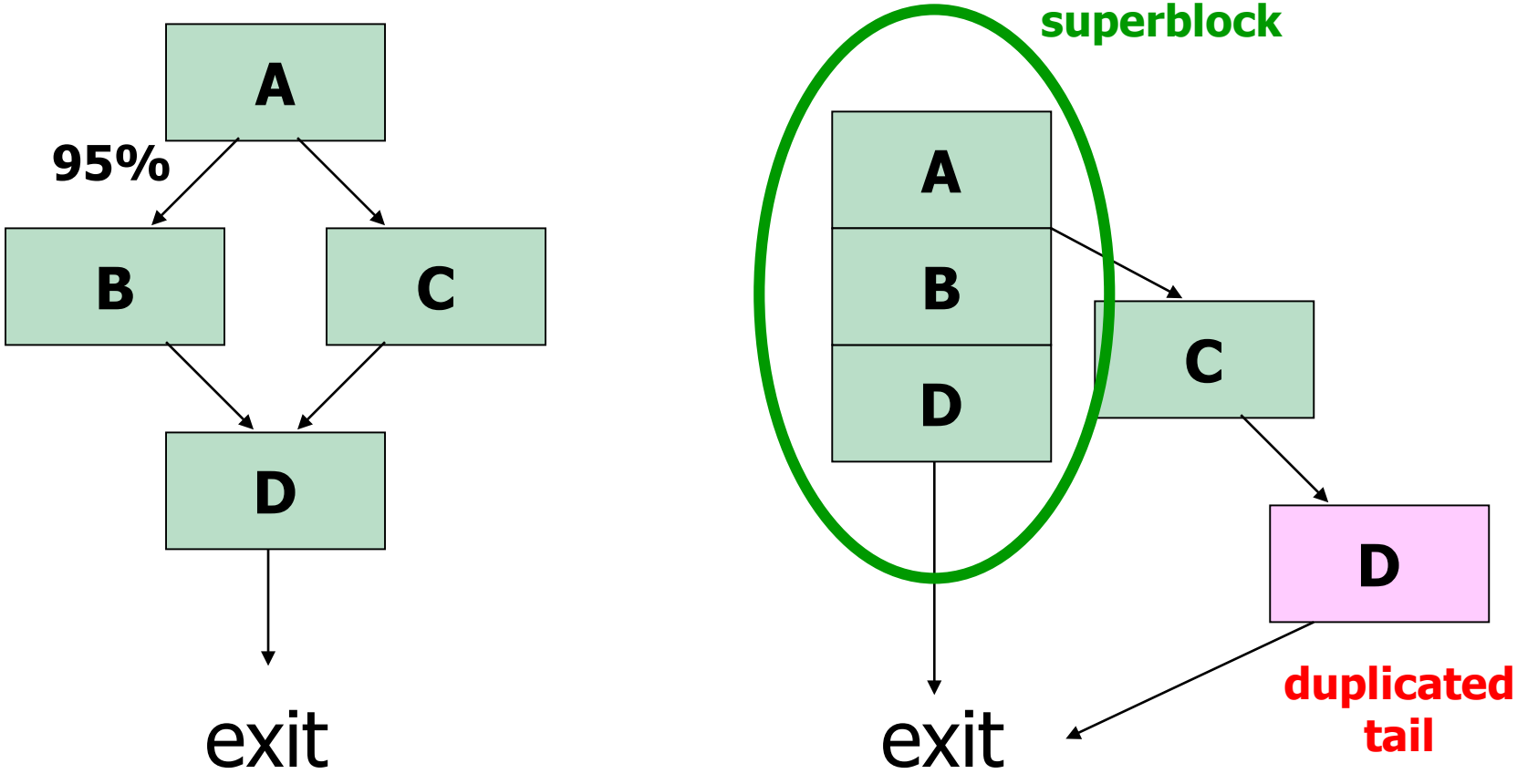
Region- (e.g., Trace-) Scheduling Is Iterative

- Generate a region (e.g., pick the most probable trace)
- Schedule it, and generate compensation code
- Now the control-data-flow graph is changed: generate again a region and schedule it again, iteratively
- Until no more compaction is possible

Beyond Trace Scheduling: Superblock Scheduling

- Extension of trace scheduling
- Moving instructions across side entrances (joins) is more expensive than moving across side exits (splits)
- Therefore → find hot traces and eliminate side entrances through tail duplication
- A **superblock** is a trace without side entrances

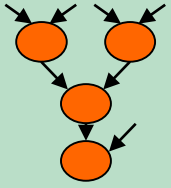
Superblock Formation



Fighting Dependencies

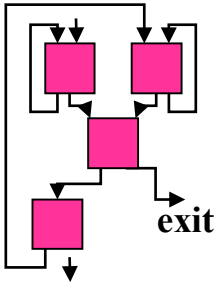
- Parallel execution is limited by the need to find **independent instructions**
- We need to deal with both **data and control** dependencies
- Data:

```
a=b+c;
d=a+d;
```

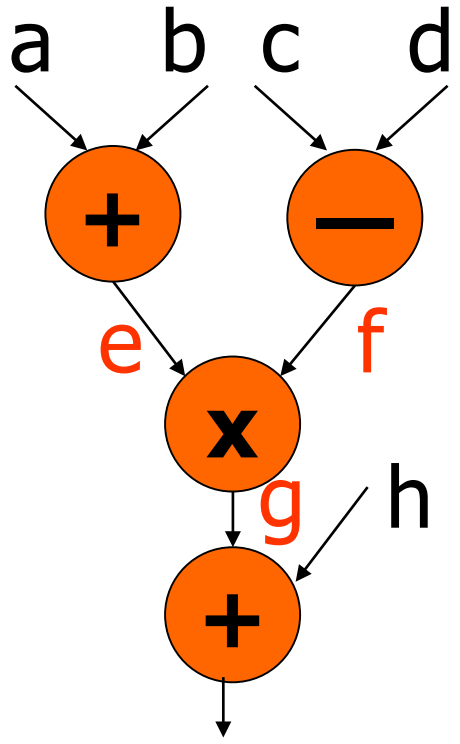


- Control:

```
if (a==b)
    d=c+d;
```



Dependencies: RAW, WAR, and WAW




```
add e a b
sub f c d
mul g e f
add i g h
```

scheduling

```
add r3 r1 r2
sub r1 r4 r5
mul r3 r3 r1
add r5 r3 r6
```

register allocation

RAW WAR WAW

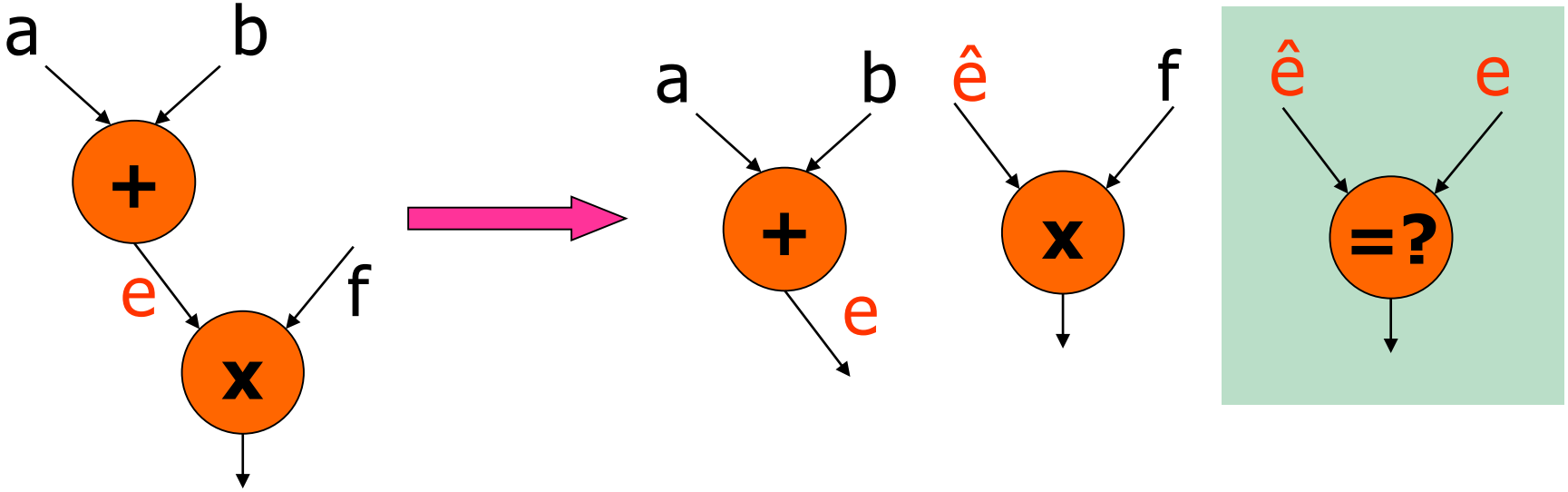
WAR and WAW are "name" dependencies...

Eliminating WAW and WAR at Compile Time

- Rename: Eliminate dependencies by using different registers at compile time
  - Need more architecturally visible registers
  - In fact, Intel's Itanium has 128 integer registers vs. 32 of typical 32-bit superscalars

RAW Dependencies Are Hard to Eliminate

- 75% of values in integer registers are predictable!



- If e is predictable, **`add`** and **`mul`** can occur in parallel (plus a comparison to verify the prediction) → Dynamically exploitable, perhaps...
- At compile time it is hard to exploit (dynamic compilation, etc.)...

Data Dependencies in Memory: Is There a Dependency?

- WAW and WAR not interesting
  - One does not want to move stores ahead of stores (WAW) or stores ahead of loads (WAR), because stores are not critical anyway

- RAW is the only important one: **moving a load above a store**
  - Can we? If same address, then there is a dependency and hence not

Example of Information Missing at Compile Time? But…

- For example, consider:

```
sw $f3, 456($r1)
lw $f0, 123($r0)
```

- Of course, we would like to start the load as early as possible (high-latency operation)
- Is there a RAW dependence?
  - At run time:
    - As soon as `$r0` and `$r1` are known, schedule freely unless

      `$r1`+456 = `$r0`+123
    - Forwarding may even hide the memory latency if RAW detected…
  - At compile time:
    - **?!…**

Memory Disambiguation at Compile Time

- At run time, we have more information on memory addresses (we have the addresses…)
- But at **compile time** we have **more time available**: we can make much more complex analyses which depend on a wider knowledge of the code

Memory Disambiguation at Compile Time

```
for i = 1 to 20 {
        j = 2 * b;
        a[2 * i + 1] = some_fn();
        b = a[j];
}
```

- Is there an integer solution to the equation

$$2i + 1 = 2b \quad ?$$

- No ($i = b - ½$) → No dependency possible
- Time consuming but possible at compile time…
- Also, other speculative techniques (assume no RAW and correct afterwards) → see later

ILP Compilation Techniques

- We have only scratched the surface with a few examples
- Many old and new techniques:
  - Aliasing analysis
  - Loop unrolling, peeling, fusion, and distribution
  - Software pipelining, modulo scheduling
  - Trace scheduling, superblock scheduling
  - With hardware support in the processor: predication, hyperblock scheduling,…
- Usually advantage **not for free**:
  - Faster only on most frequent part of the code; penalty elsewhere → need a good static prediction of execution frequencies
  - Difficult to apply some techniques in the general case
  - Somehow larger code (e.g., worsens the performance of the I-cache…)

Conclusions on VLIW Compilers

- Many **difficult** decisions
  - Which type of region is right? Traces, superblocks, hyperblocks, treegions?
  - Which regions to optimise?
    - Can one ask users to profile their code?
    - Can one compile without profiling information?
  - To unroll or not to unroll? How many times?
  - To predicate or not to predicate?
  - When to allocate registers? (e.g., before, during, or after scheduling)

➔ Powerful compiler backends for VLIWs are **very hard** to build

5

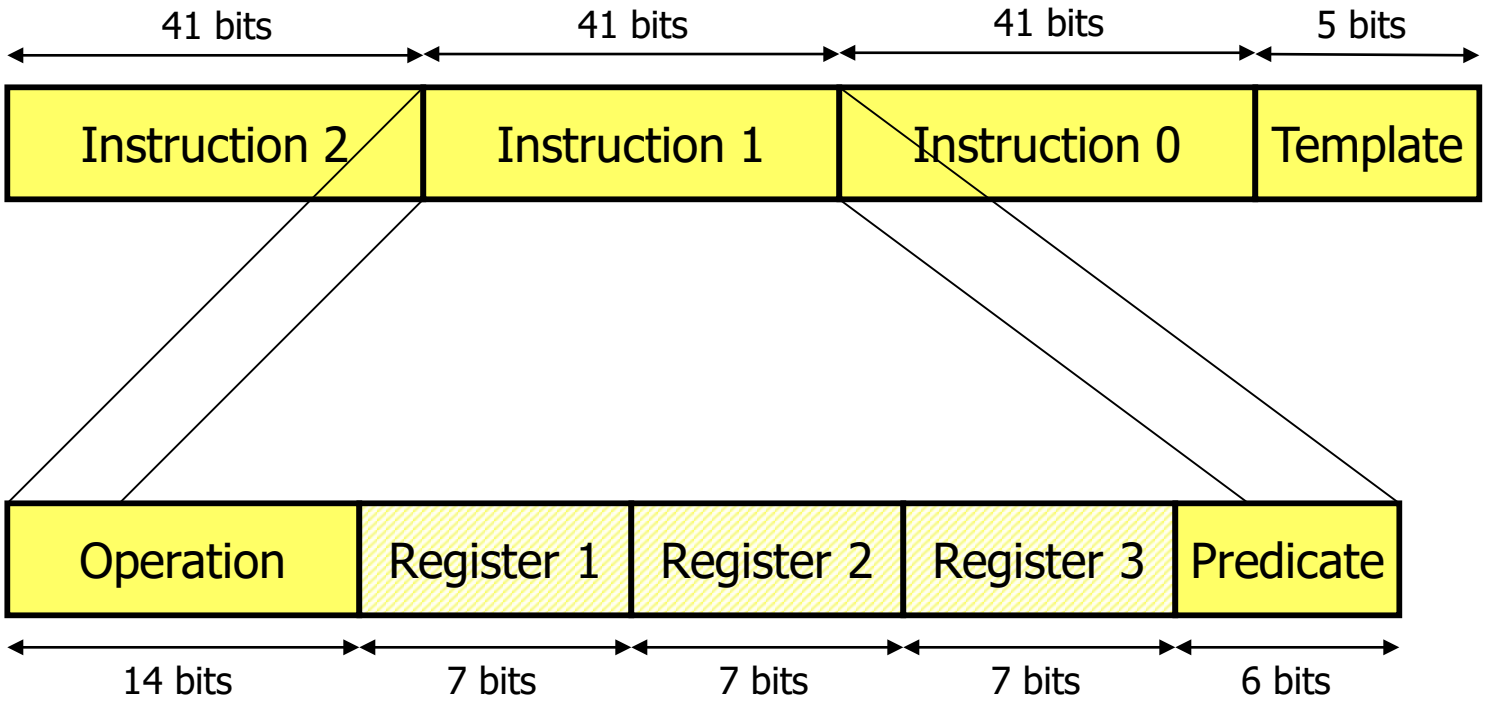
IA-64 and Itanium 2

(A Real VLIW Processor?!...)

What is IA-64? What is Itanium?

- In December 1993, HP and Intel started discussing cooperation on high-end processors
- In June 1994, HP and Intel announced a partnership to develop a completely new 64-bit *EPIC (Explicit Parallel Instruction Computing)* architecture
- VLIW-related ideas come from HP Labs, some pieces of compiler technology from the Impact Group at University of Illinois
- Intel started implementing the IA-64 architecture (Itanium)
- The first Itanium-based systems appeared mid-2001
- Itanium 2 processor (McKinley) was released in 2002 and discontinued in 2007; other implementations followed until 2017
- Itanium 2 was the largest area and largest transistor-count processor ever arrived on the market
- HP and Intel have poured significant investment (1 billion USD?) in the IA-64 architecture; they sold about 55k units in 2007 vs. a market of 8.4M x86 units
- Itanium architecture reached the official end of life in 2021

EPIC 128-bit Instruction Bundles



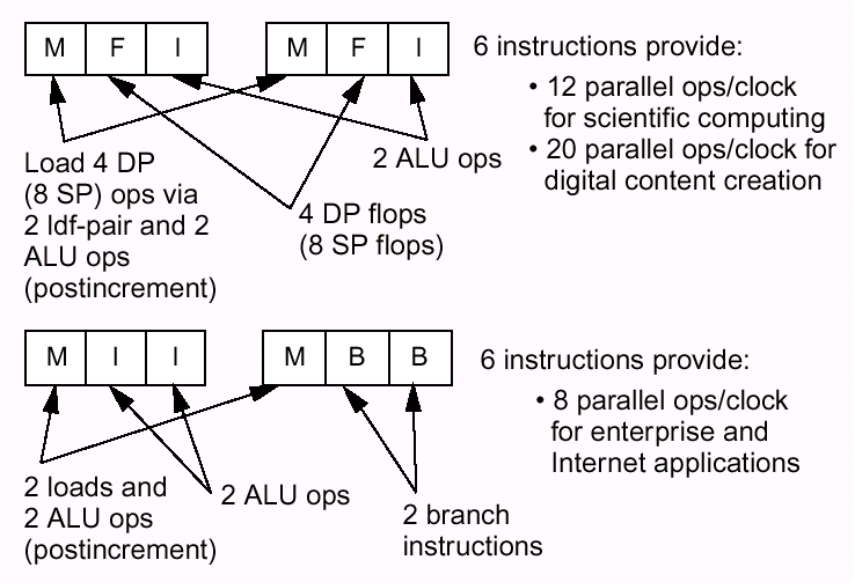
Bundles
Fetch and Execution Packets

```
{.mii
add r1 = r2, r3
sub r4 = r4, r5 ;;
shr r7 = r4, r12 ;;}
{.mmi
ld8 r2 = [r1];;
st8 [r1] = r23
tbit p1, p2 = r4, 5}
{.mbb
ld8 r45 = [r55]
(p3)br.call b1 = func1
(p4)br.cond Label1}
{.mfi
st4 [r45] = r6
fmac f1 = f2, f3
add r3 = r3, 8 ;;}
```

1st, 2nd, 3rd, 4th — Bundles

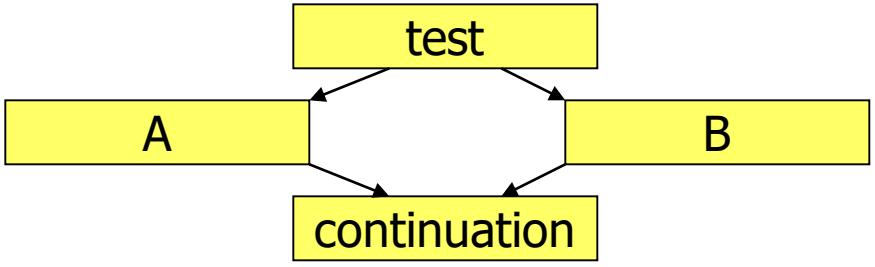
1st, 2nd, 3rd, 4th — Cycles

Statically defined delivery through **templates**



24 templates define different combinations of delivery and stop bits

Predication (I)




```
if (test) {
        code A}
else {
        code B};
```

- Frequent **sequence** for poorly predictable branches
  1. Conditional branch (e.g., if r1 == r2)
  2. Speculative instructions executed
  3. Branch resolved (misprediction)
  4. Speculative instructions squashed
  5. Correct instructions executed
- How to reduce the cost due to the sequential execution 2-3-4? Is it possible to avoid the Branch altogether?

Predication (II)

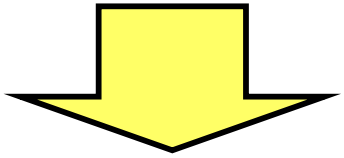
- Practically every instruction can be executed conditionally depending on the value of a Boolean register (predicate)
- Special instructions set a predicate as a result of comparisons and tests
- Example

```
        cmp.eq p1, p2 = r1, r2;;     // p1 = (r1==r2)
                                     // p2 = !p1
(p1)    sub r9 = r10, r11            // if (p1) sub…
(p2)    add r5 = r6, r7              // if (!p1) add…
```

- Both control paths are executed simultaneously—no more branch/jumps

Compound AND/OR for Predication

```
if ((a==0) || (b<=5) || (c!=d) || (f&0x2)) {
      r3 = 8;
}
```



```
cmp.ne p1 = r0, r0          // p1 = false
add t = -5, b;;             // t = b - 5

cmp.eq.or p1 = 0, a         // p1 = p1 || (a==0)
cmp.ge.or p1 = 0, t         // p1 = p1 || (t<=0)
cmp.ne.or p1 = c, d         // p1 = p1 || (c!=d)
tbit.or p1 = 1, f, 1;;      // p1 = p1 || (f&0x2)

(p1) mov r3 = 8             // if (p1) r3 = 8
```

Single cycle

Multiway Branches Through Predication

- Often code contains sequences of branches (e.g., **`switch`** in C) which would be useful to execute in parallel
- Multiway branches:

```
{.mii
        cmp.eq p1 = r1, r2      // p1 = (r1==r2)
        cmp.ne p2 = 4, r5       // p2 = (r5!=4)
        cmp.lt p3 = r8, r9}     // p3 = (r8<r9)
{.bbb
  (p1)  br.cond label1          // if (p1) goto label1
  (p2)  br.cond label2          // else if (p2) goto label2
  (p3)  br.call b4 = label3}    // else if (p3) label3()
```

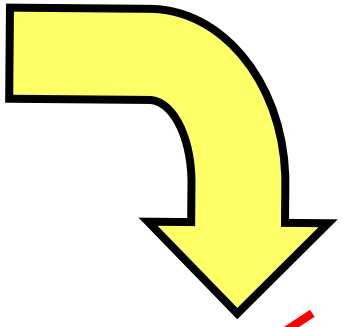
Balance Between Static and Dynamic Branch Prediction

- Predication reduces number of branches
- Hardware support in Itanium for prediction
  - Two direction prediction tables
  - Several target prediction schemes
- Many types of branch **hints from compiler**
  - Use static only prediction (save table space)
  - Taken/Not Taken (static or default value)
  - Deallocate space in tables
  - Prefetch hints (no prefetch, few lines, many lines)
  - Branch Prepare instruction

Control Speculation (I)

- **Goal**: move loads as early as possible, even **speculatively** before preceding branches (i.e., without being sure that they are really needed)

```
<some code>
(p1) br.cond somewhere
// ------ barrier
ld r1 = [r2]
<some code using r1>
```




```
ld r1 = [r2]

<some code>
(p1) br.cond somewhere
// ------ barrier
<some code using r1>
```

```
// load could be speculated
// if old value r1 not needed
// <- neither here nor
//    in "somewhere"

// but…
```

Control Speculation (II)

- Speculative loads must not raise "speculative" (false) exceptions, thus **deferred exceptions**

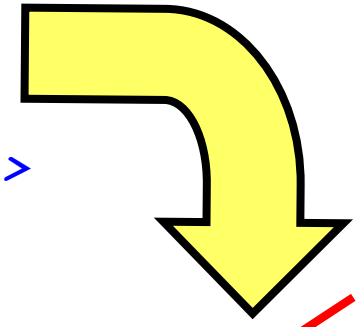
```
ld.s r1 = [r2]              // speculative loads do not raise
                            // exceptions but mark the register
                            // with the additional NaT bit
<some code>
<some code using r1>        // NaT is propagated in further
                            // calculations, which also
                            // defer exceptions
(p1) br.cond somewhere
// ------ barrier
<some more code using r1>
chk.s r1, fix_code_r1       // call exception handler if needed
                            // to fix-up execution
```

- Important advantage because loads (slow operations) can now be started earlier

Data Speculation (I)

- Similarly, potential RAW dependencies through memory are to be conservatively assumed as real dependencies → Loss of useful reordering possibilities
- **Goal**: move loads as early as possible, even **speculatively** before preceding stores (i.e., without being sure that the value is right)

```
<some code>
st [r3] = r4
// ------ barrier
ld r1 = [r2]
<some code using r1>
```




```
ld r1 = [r2]          // load could be speculated…
<some code>
st [r3] = r4          // …but if r2==r3, r1 is WRONG!
// ------ barrier
<some code using r1>
```

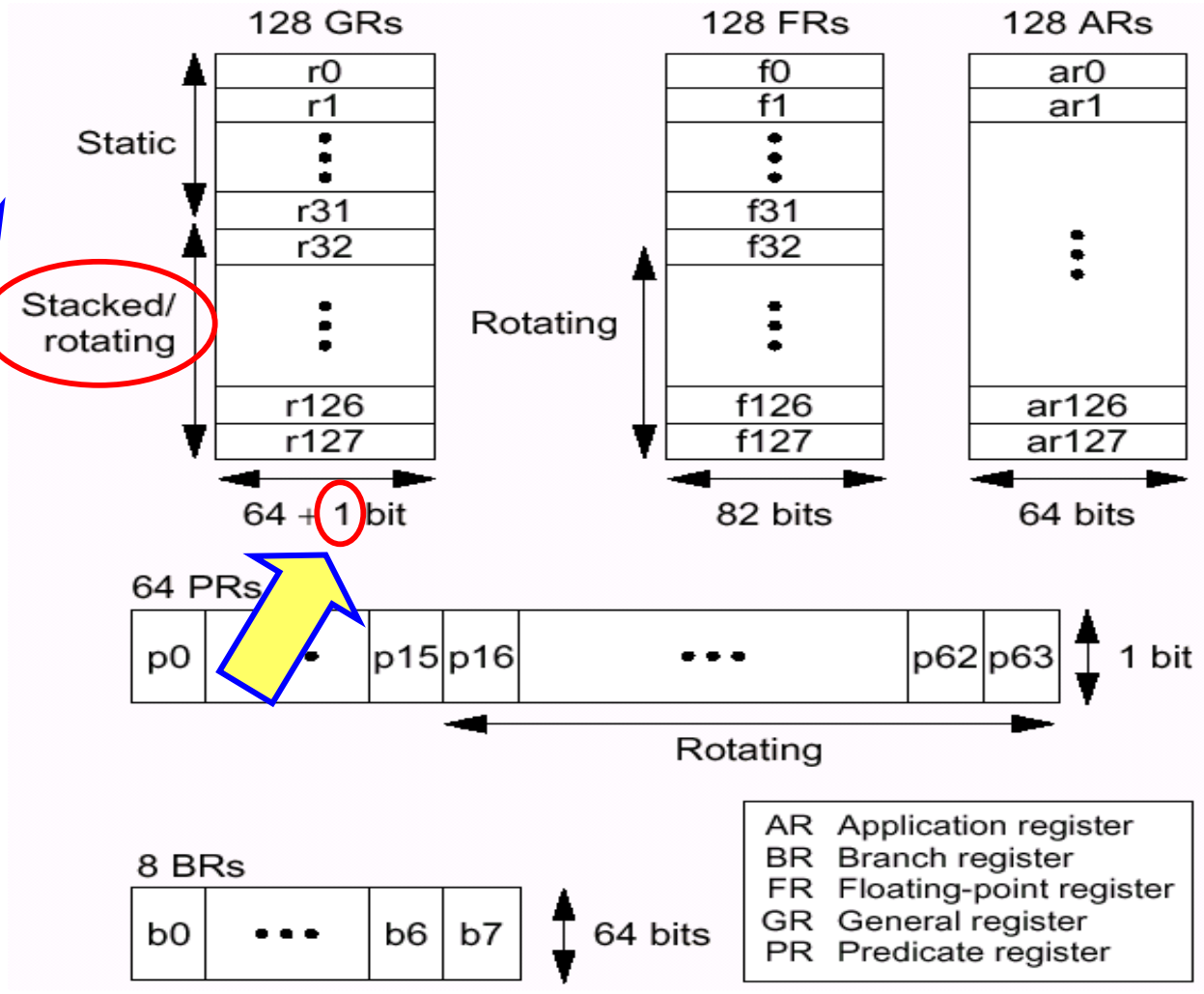
Data Speculation (II)

- Speculative Loads get executed but mark the destination register as "speculatively" loaded and track subsequent stores for a conflict

```
ld.a r1 = [r2]              // speculative loads are normal
                            // but mark always the register
                            // with the additional NaT bit
<some code>
<some code using r1>        // NaT is propagated in further
                            // calculations

st [r3] = r4                // successive stores are checked
                            // to see if they rewrite locations
                            // which were object of speculative
                            // loads
// ------ barrier
<some more code using r1>
chk.a r1, fix_code_r1       // if violated RAW dependence, call
                            // special fix-up routine
```

- Important advantage because loads (slow operations) can now be started earlier

Application State — Registers



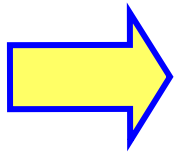
Register Model (I)
Stacked Registers

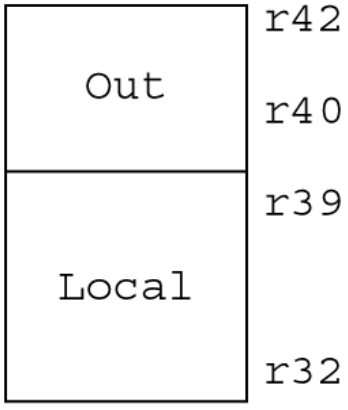
- Registers #0-31 are static (normal registers)
- Each procedure sees a fresh register set from #31 onwards (max 96)
- Special instruction

```
alloc <local-regs>, <out-regs>
```

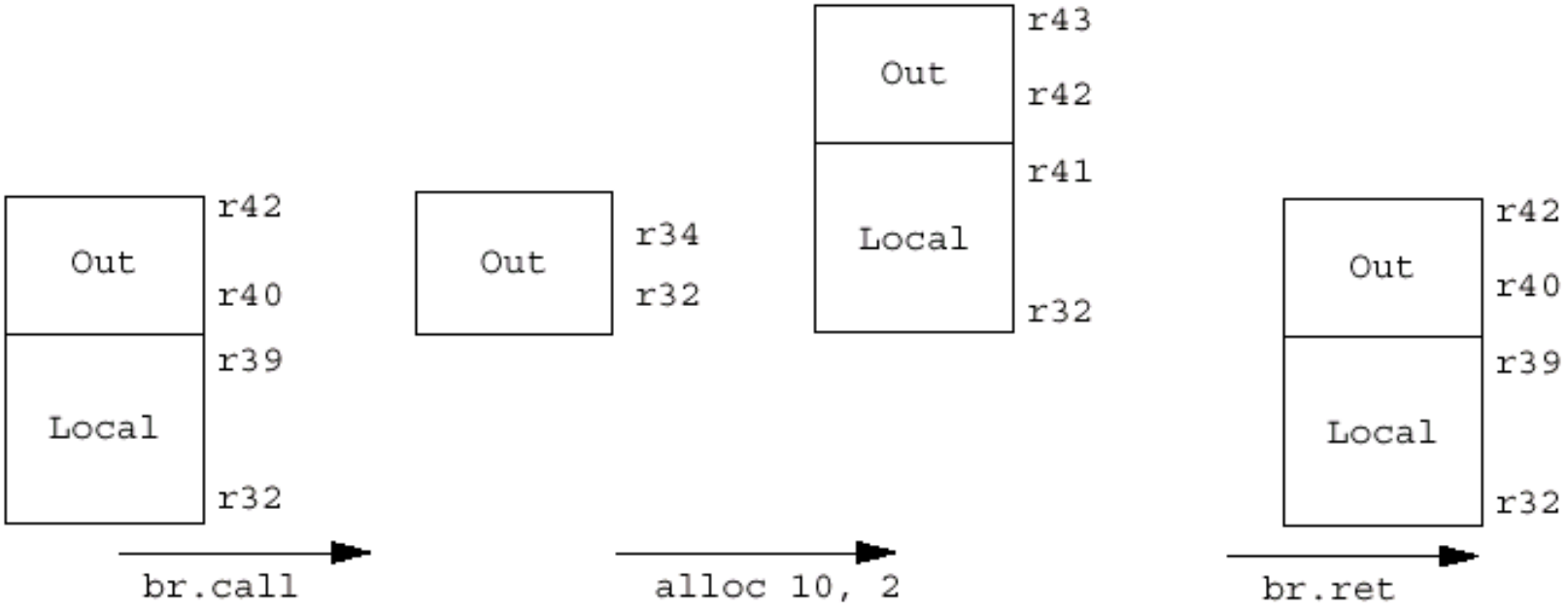
- Declares max number of registers used in a procedure and max number of registers passed to a called procedure

```
alloc 8, 3
```





Register Model (I)
Stacked Registers



The calling procedure has 8 local and 3 output registers

After a call, only the 3 output registers are visible to the called routine

After an **`alloc`**, the old output registers are part of the new 10 local registers

After a return, everything goes back as before the call

Register Model (I)
Stacked Registers

- Addresses the fact that parameter exchange through the stack before and after a function call (arguments and result) is very expensive if memory is a bottleneck (think also of registers `$a0`-`$a3` and `$v0`-`$v1` in MIPS)
- The basic idea of is very similar to Register Windows in the SPARC architecture but more flexible:
  - SPARC has 128 registers R0-R127 but only 32 are visible at once
  - r0-r7 = R0-R7 are Globals and always visible
  - r8-r31 are a window (initially r31 = R127)
    - r8-r15 = out,
    - r16-r23 = locals, and
    - r24-r31 = in
  - At each CALL, the active window is moved down 16 registers so that r8-r15 (outs of the previous procedure) become r24-r31 (ins of the new procedure) and all other registers are fresh

Register Model (I)
Stacked Registers

- What happens if one `alloc`'s more registers than physically available?
  - Number and type (# in/outs regs) of nested calls is **dynamic**!
  - SPARC generates an exception
  - In Itanium, a Register Stack Engine spills registers of outer procedures (oldest in the stack)
- Asynchronous and autonomous spilling of the non-visible registers in the background
  - Can do spilling speculatively ahead of time
  - Tries to use free Load/Store slots
  - Reported effectiveness: removes 30% of Loads/Stores and consumes only 5% of the execution slots
- One step further in dynamic speculative execution!
  Weren't VLIW "static" processors?!...

Register Model (II) — Rotating Registers and Software Pipelining

- **Loop unrolling** and **Software Pipelining** (see before) are ways to achieve more ILP in small loop bodies—but both have a number of tangible limitations (e.g., larger code, limited applicability)
- **Modulo Scheduling** achieves the same purpose more effectively
  - **Rotating registers** and **Predicates** are the microarchitectural support needed to implement these techniques

Reminder: SW Pipelining Example

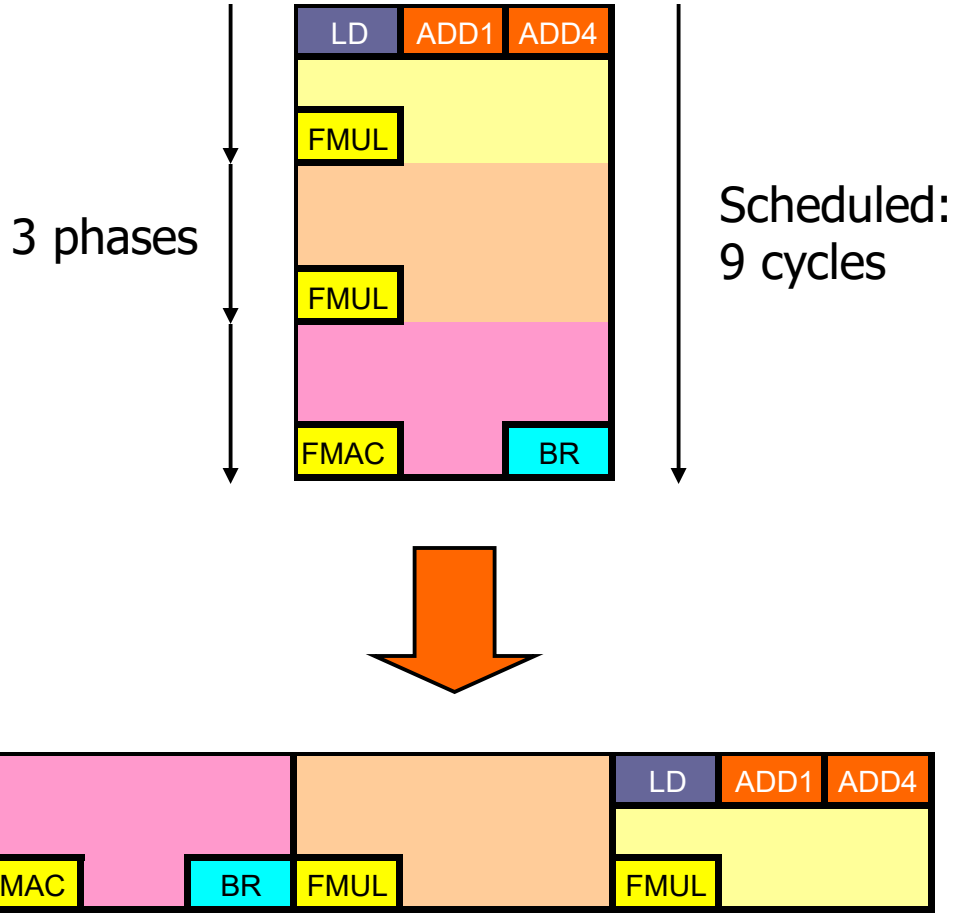
LOAD /Branch Unit	STORE Unit	ALU	Floating-Point Unit
`ld $f0, ($r1)`			
`ld $f6, -8($r1)`			
`ld $f0, -16($r1)`			`addd $f4,$f0,$f2`
`ld $f6, -24($r1)`			`addd $f8,$f6,$f2`
`ld $f0, -32($r1)`			`addd $f12,$f0,$f2`
`ld $f6, -40($r1)`	`sd 0($r1), $f4`		`addd $f4,$f10,$f2`
`ld $f10, -48($r1)`	`sd -8($r1),$f8`		`addd $f8,$f14,$f2`
`ld $f14, -56($r1)`	`sd -16($r1),$f12`	`subi $r1,$r1,24`	`addd $f12,$f6,$f2`
`bnez $r1, Loop`			
	`sd 0($r1), $f4`		`addd $f4,$f10,$f2`
	`sd -8($r1), $f8`		`addd $f8,$f14,$f2`
	`sd -16($r1),$f12`		
	`sd -24($r1), $f4`		
	`sd -32($r1), $f8`		

Modulo Scheduling

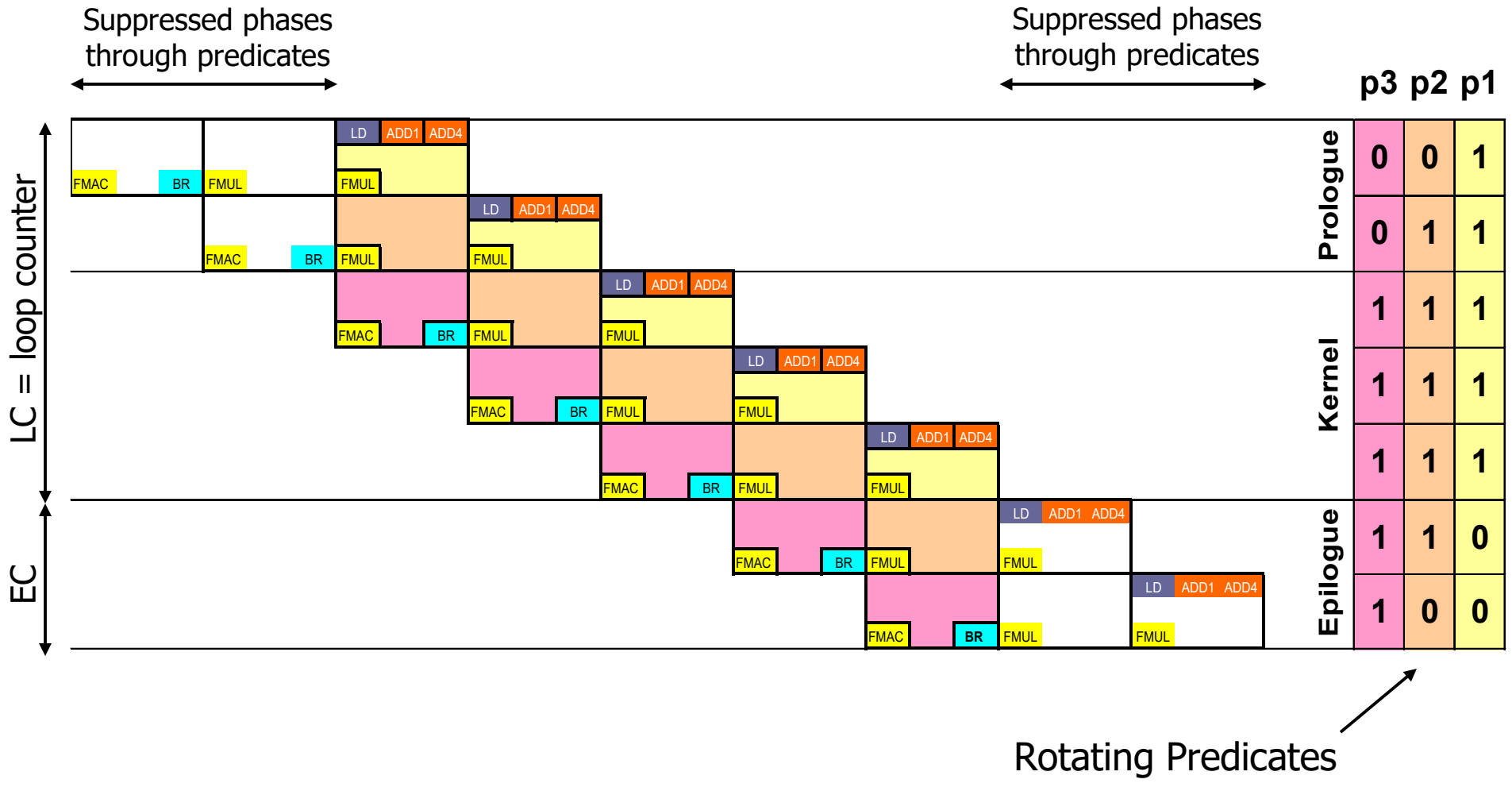
- Goals:
  - Get rid of the **Prologue** and **Epilogue** → use the loop **Kernel** instead
  - Minimize size of the **Kernel**
  - Automate/hide loop counting
- Solution:
  - Architectural "renaming" across iterations → **Register Rotation**
    - Every new iteration r32→r33, r33→r34, r34→r35, etc.
  - Special use of the **predicates and loop instructions** to mask out instructions in the prologue and epilogue

Software Pipelining Reminder: Restructuring of the Loop Kernel

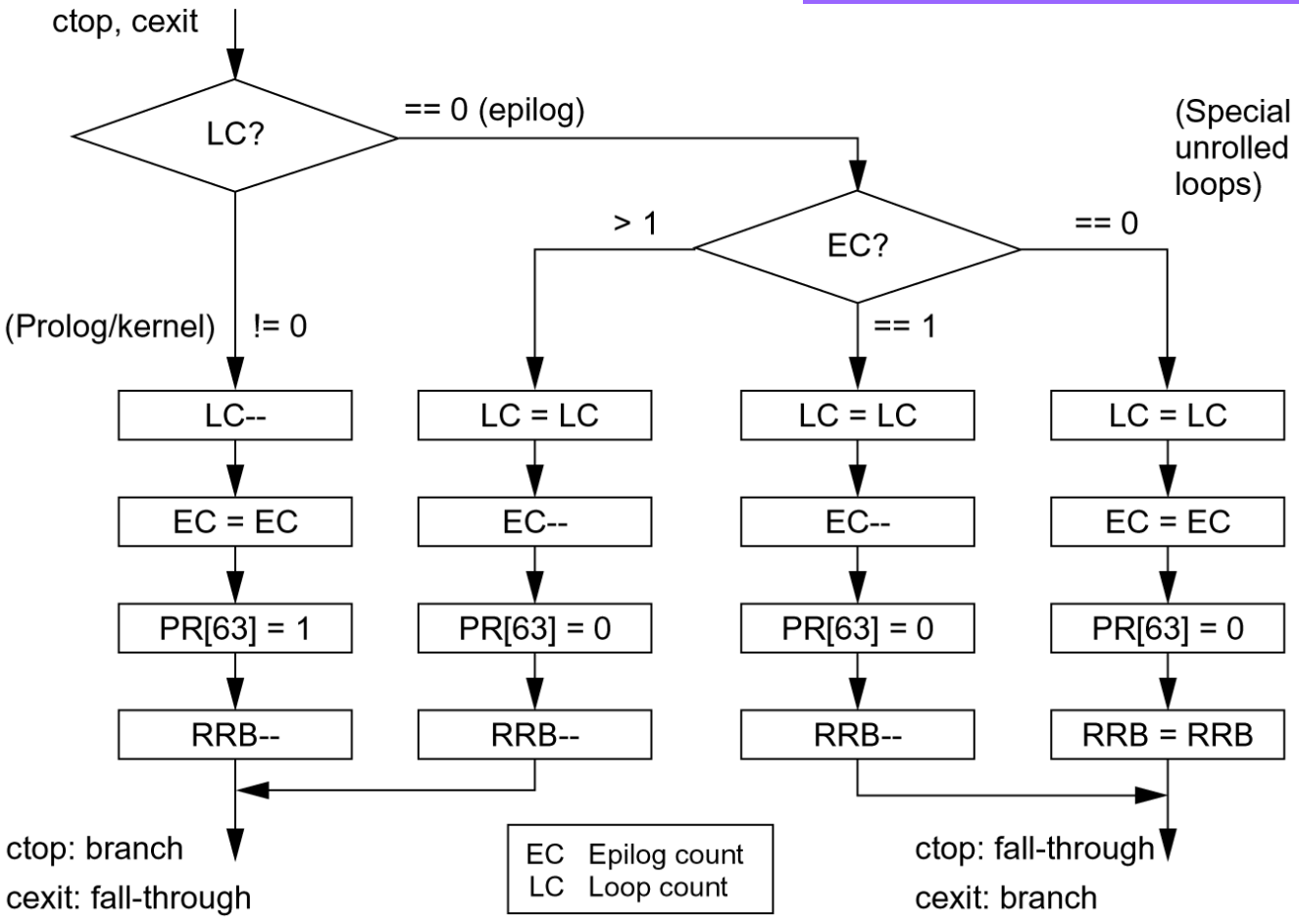
- Restructuring of the loop kernel from "vertical" (sequential) to "horizontal" (parallel)
- Parallelism among different iterations



Modulo Scheduling



Register Model (II) — Rotating Registers and Loop-Type Branches



Special actions on:

- **Loop Counter**
  - Count iterations (prologue and kernel)
- **Epilogue Counter**
  - Count epilogue iterations
- **Predicates**
  - mask out epilogue
- **Rotate** all registers (r32→r33, r33→r34, etc.) incl. predicates

98 7-bit adders and **42 MUXes** to implement RFs stacking and rotations

Example of Modulo Scheduling

- Add a constant to a vector

```
        mov LC = 99                    // LC = loop trip count - 1
        mov EC = 4                     // EC = epilogue stages + 1
        mov pr.rot = 1 << 16           // p16 = 1, rest = 0
Loop:  (p16) ld4 r32 = [r5], 4
       (p18) add r35 = r34, r9
       (p19) st4 [r6] = r36, 4
       br.ctop Loop ;;
```

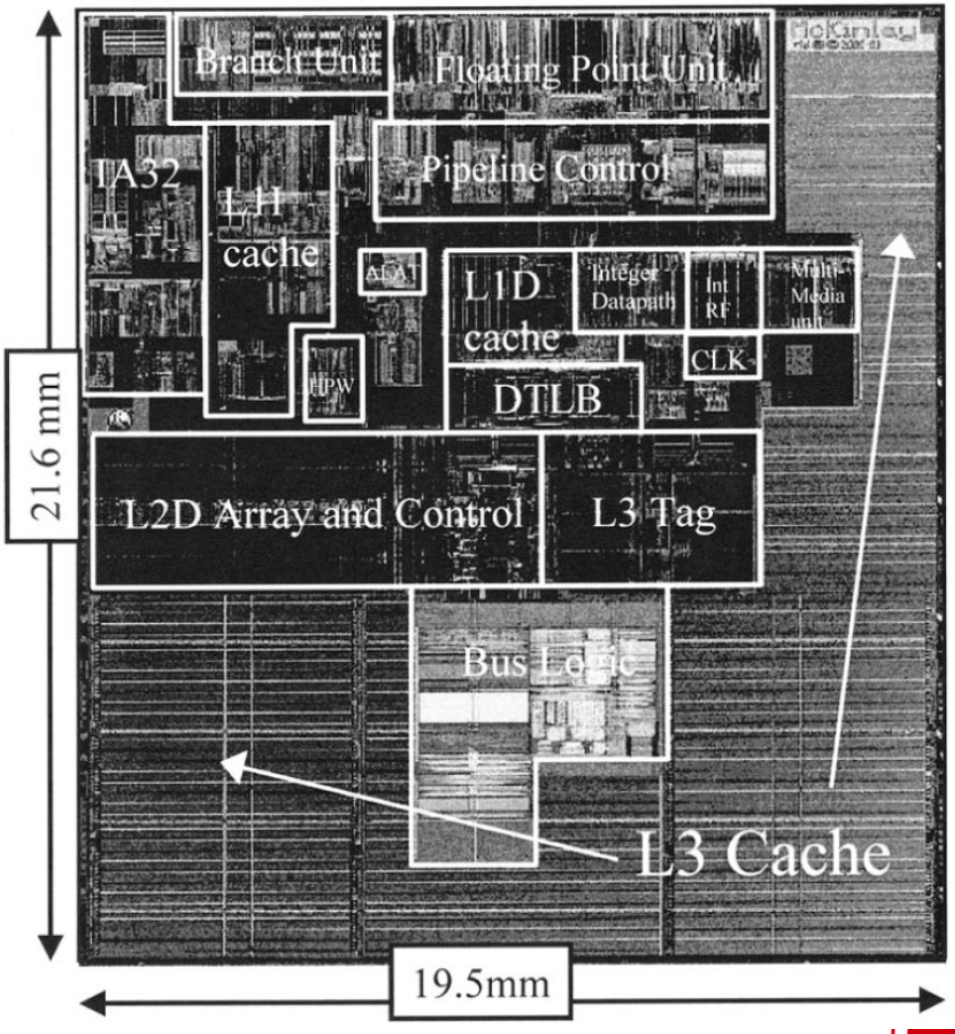
- Remarks:
  - `p16` to `p19` in the loop: four phases in a single VLIW instruction
  - Second phase empty
  - `ld4` has 2-cycle latency, hence `r34` is the result of `ld4`
  - `add` has 1-cycle latency, hence `r36` is the result of `add`
  - The immediate 4 in `ld4` and `st4` is post added to the memory pointer

Miscellaneous Features

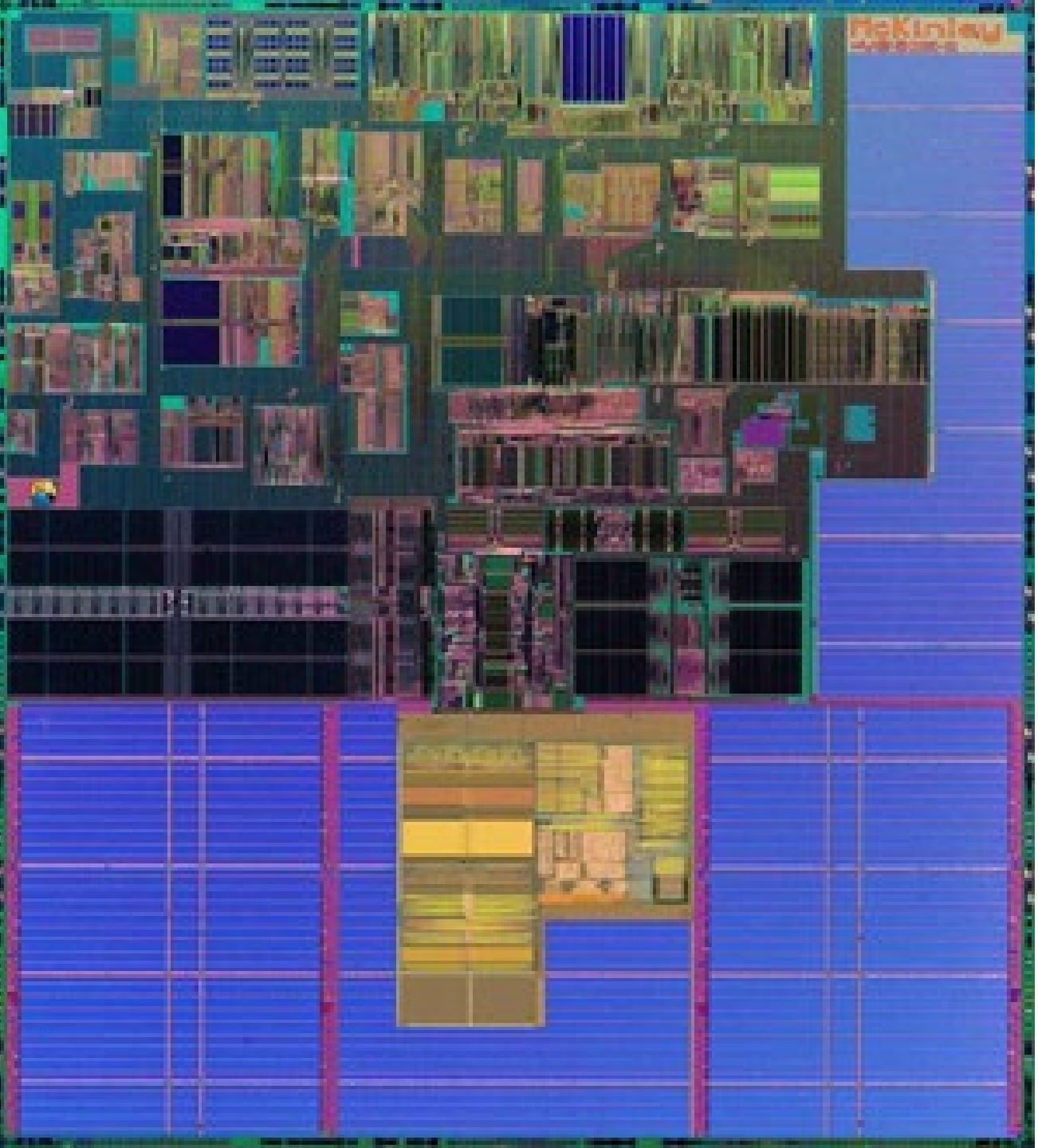
- Very large Virtual Memory Model
  - Support for 64-bit addresses = 16 billion GBytes
- 82-bit Floating Point support
  - 32-bit Single Precision IEEE-754
  - 64-bit Double Precision IEEE-754
  - 80-bit Double-extended Precision IEEE-754
    - Two additional bits to increase efficiency
  - 2 x 32-bit Single Precision IEEE-754 (SIMD)

Itanium 2 Chip

- Second commercial implementation of IA-64
- 1GHz in a .18um CMOS 6M process
- 8-stage pipeline
- Issues up to 8 instructions per cycle on 19 (?) execution units
- 16Kb+16Kb L1 Data and Instruction caches
- 256Kb L2 unified cache
- 3Mb **L3 on-chip** unified cache
- 128-bit data bus, sustaining 400Mbit/s/pin → 6.4 Gbit/s
- **Huge** die:
  - 400mm^2
  - 221M transistors



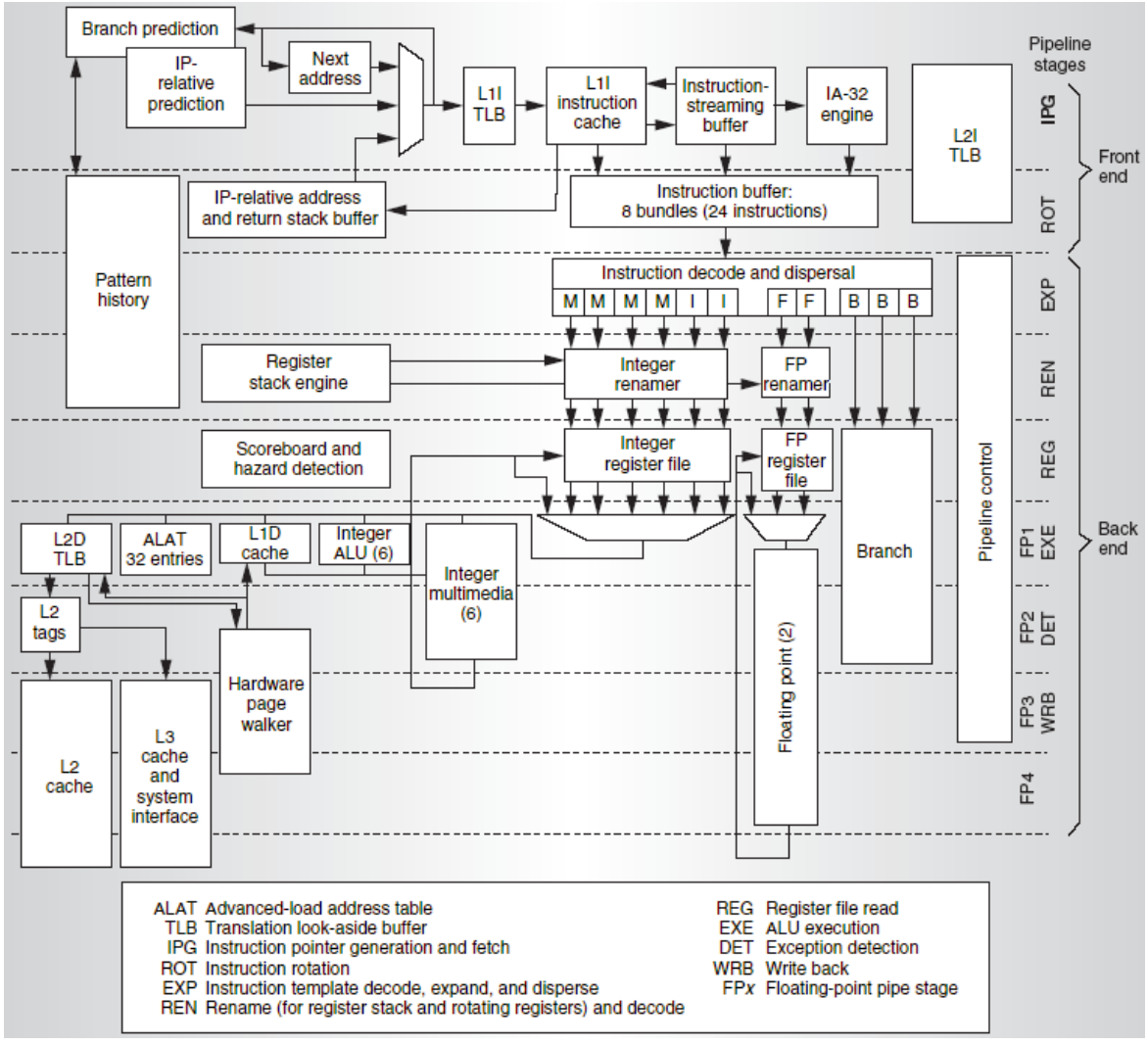
Source: Naffziger et al., © IEEE 2002



Itanium 2 Chip

Source: Microprocessor Report, © Cahners 2002

Itanium 2 Processor Pipeline



Source: McNairy and Soltis, © IEEE 2003

Source: Microprocessor Report, © Cahners 2009

Processor	Intel 1-core Xeon	AMD 1-core Opteron 854	Intel 2-core Xeon X5270 [1]	AMD 2-core Opteron 8224SE	Intel 4-core Xeon X7350 [2]	AMD 4-core Opteron 8360SE [3]	Intel 6-core Xeon X7460 [4]
Bit-width	32/64-bit	32/64-bit	32/64-bit	32/64-bit	32/64-bit	32/64-bit	32/64-bit
Cores/chip x Threads/core	1 x 2	1 x 1	2 x 1	2 x 1	4 x 1	4 x 1	6 x 1
Clock Rate	3.80GHz	2.80GHz	3.50GHz	3.20GHz	2.93GHz	2.50GHz	2.67GHz
Cache: L1-L2-L3 - I/D or Unified	12K/16K - 2M - N/A	64K/64K - 1M - N/A	2 x 32K/32K - 6M - NA	2 x 64K/64K - 2 x 1M - N/A	4 x 32K/32K - 2 x 4M - N/A	4 x 64K/64K - 4 x 512K - 2M	6 x 32K/32K - 3 x 3M - 16M
Execution Rate/Core	3 Instructions	3 Instructions	1 Complex + 3 Simple	3 Instructions	1 Complex + 3 Simple	3 Instructions	1 Complex + 3 Simple
Pipeline Stages	31	12 int / 17 fp	14	12 int / 17 fp	14	12 int / 17 fp	14
Out of Order	126	72	96	72	96	72	96
Memory Bus	800MHz	6.4GB/s	1333MHz	10.6GB/s	1066MHz	10.6GB/s	1064MHz
Package	LGA-775	uPGA 940	LGA-771	LGA-1207	LGA-771	LGA-1207	LGA-771
IC Process	90nm 7M	90nm 9M	45nm	90nm 9M	65nm 8M	65nm 11M	45nm
Die Size	109mm^2	106 mm^2	107mm^2	227mm^2	2 x 143mm^2	283mm^2	503mm^2
Transistors	169M	120M	410M	233M	2 x 291M	463M	1900M
List Price (Intro)	$903	$1,514	$1,172	$2,149	$2,301	$2,149	$2,729
Power (Max)	110W	93W	80W	120W	130W	105W	130W
Availability	3Q05	3Q05	3Q08	3Q07	3Q07	2Q08	4Q08
Scalability	1–2 Chips	2–4 Chips	1–2 Chips	1–4 Chips	1–4 Chips	2–4 Chips	1–4 Chips
SPECint/fp2006 [Cores]	11.4/11.7 [2]	11.2/12.1 [2]	26.5^x/25.5^x [4]	14.1/14.2 [8]	21.7^y/18.9^y [16]	14.4^x/18.5^x [8]	22.0^x/22.3^x [24]
SPECint/fp2006_rate [Cores]	20.9/18.8 [2]	41.4/45.6 [4]	84.9^x/57.7^x [4]	105/96.7 [8]	184^y/108 [16]	170^x/156^x [16]	274^x/142^x [24]
x86 Codename	Irwindale	Athens	Wolfdale	Santa Rosa	Tigerton	Barcelona	Dunnington
Microarchitecture	Netburst	K8	Core	K8	Core	K10	Core

Processor	Intel Itanium 2 9050	Intel Itanium 9150M	IBM POWER5+	IBM POWER6	Fujitsu SPARC64 VI	Fujitsu SPARC64 VII	Sun UltraSPARC T2+
Bit-width	64-bit	64-bit	64-bit	64-bit	64-bit	64-bit	64-bit
Cores/Chip x Threads/Core	2 x 2	2 x 2	2 x 2	2 x 2	2 x 2	4 x 2	8 x 8
Clock Rate	1.60GHz	1.67GHz	2.20GHz	5.00GHz	2.40GHz	2.52GHz	1.40GHz
Cache: L1-L2-L3 - I/D or Unified	2 x 16K/16K - 1M/256K - 12M(on)	2 x 16K/16K - 1M/256K - 12M(on)	2 x 64K/32K 1.92M - 36M(off)	2 x 64K/64K - 2 x 4M - 32M(off)	2 x 128K/128K - 6M - N/A	4 x 64K/64K - 6M - N/A	8 x 8K/16K - 4M - NA
Execution Rate/Core	6 Issue	6 Issue	5 Issue	7 Issue	4 Issue	4 Issue	16 Issue
Pipeline Stages	8	8	15	13	15	15	8 int / 12 fp
Out of Order	None	None	200	Limited	64	64	None
Memory Bus	8.5GB/s	10.6GB/s	12.8GB/s	75GB/s	8GB/s	8GB/s	42.7GB/s
Package	mPGA-700	mPGA-700	MCM-5370 Pins	N/A	412 I/O Pins	412 I/O Pins	1831 Pins
IC Process	90nm 7M	90nm 7M	90nm 10M	65nm 10M	90nm 10M	65nm 11M	65nm
Die Size	596mm^2	596mm^2	245mm^2	341mm^2	421mm^2	400mm^2	342mm^2
Transistors	1.72B	1.72B	276M	790M	540M	600M	503M
List Price (Intro)	$3,692	$3,692	N/A	N/A	N/A	N/A	N/A
Power (Max)	104W	104W	100W	>100W	120W	135W	95W
Availability	3Q06	4Q07	4Q05	2Q08	2Q07	3Q08	2Q08
Scalability	1–64 Chips	8–128 Chips	1–32 Chips	2–32 Chips	4–64 Chips	4–64 Chips	2 Chips
SPECint/fp2006 [Cores]	14.5/17.3 [2]	N/A	10.5/12.9 [1]	15.8*/20.1 [1]	9.7/21.7^y [32]	10.5^x/25.0^x [64]	N/A
SPECint/fp2006_rate [Cores]	1534/1671 [128]	2893/N/A [256]	197/229 [16]	1837^x/1822 [64]	1111/1160 [128]	2088^x/1861^x [256]	142/111 [16]
Architecture Status	Inactive	Active	Inactive	Active	Inactive	Active	Active

All SPEC scores are base. * Score measured at 4.20GHz (not 5.00GHz).

Current High-End Processors

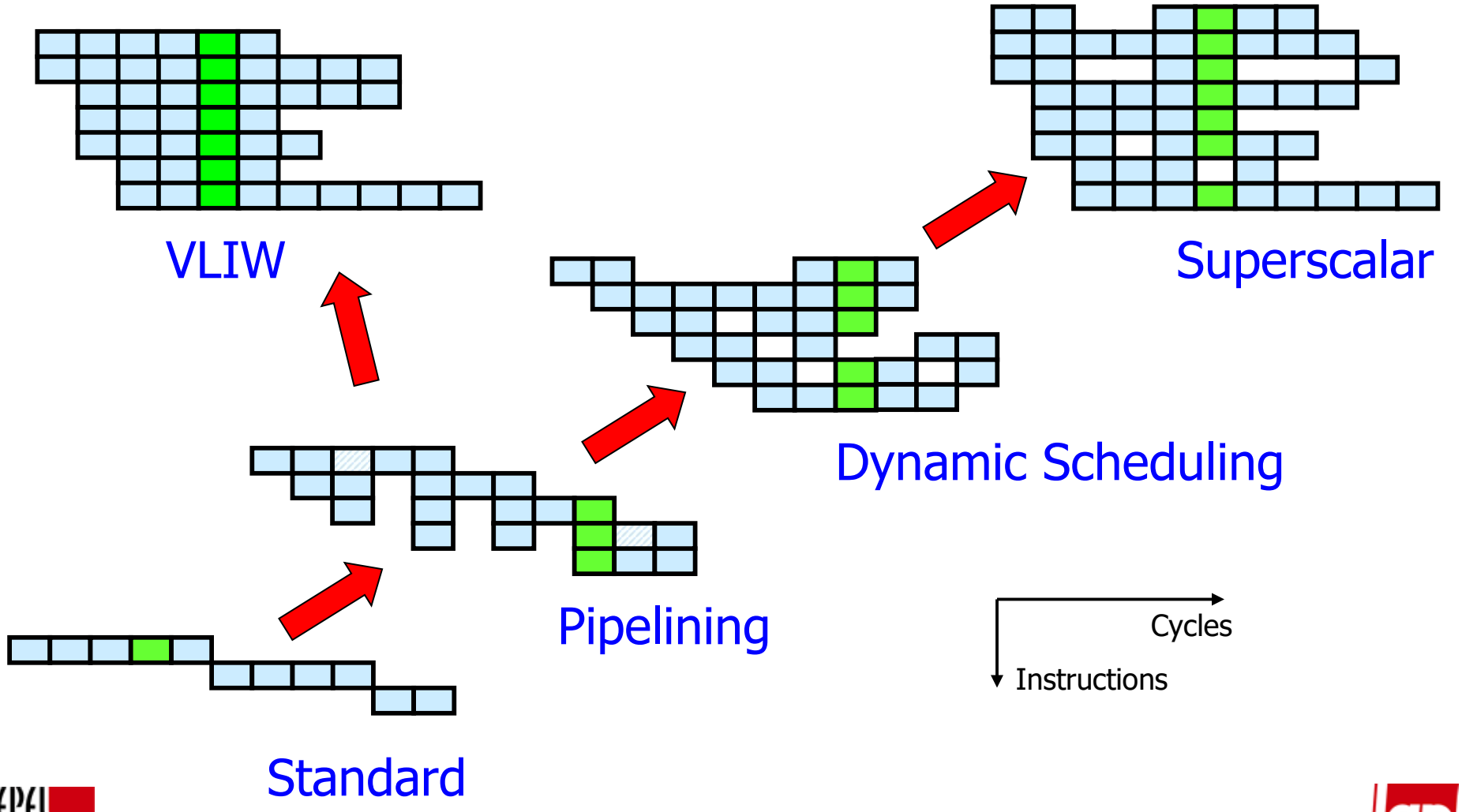
It works!...

But if one compares Itanium 2 and 1-core Xeon (same technology), Itanium 2 has slightly better performance (+30-50%) at the price of **~6 times larger area** and **~10 times more transistors**...

But! Mutual Exclusion of Static and Dynamic Scheduling? No...

- Itanium (IA-64) code is EPIC—that is, it is **statically scheduled** in 3-instruction 128-bit bundles
- Merced (2001) and McKinley (2002) **issue in order** 2 bundles in parallel
- The business importance of binary compatibility, will **possibly make future implementation of IA-64 dynamically scheduled** sometimes in the (not-too-near?) future

Two Ways to ILP
Both Available in High-End Systems



Conclusions on Real VLIWs

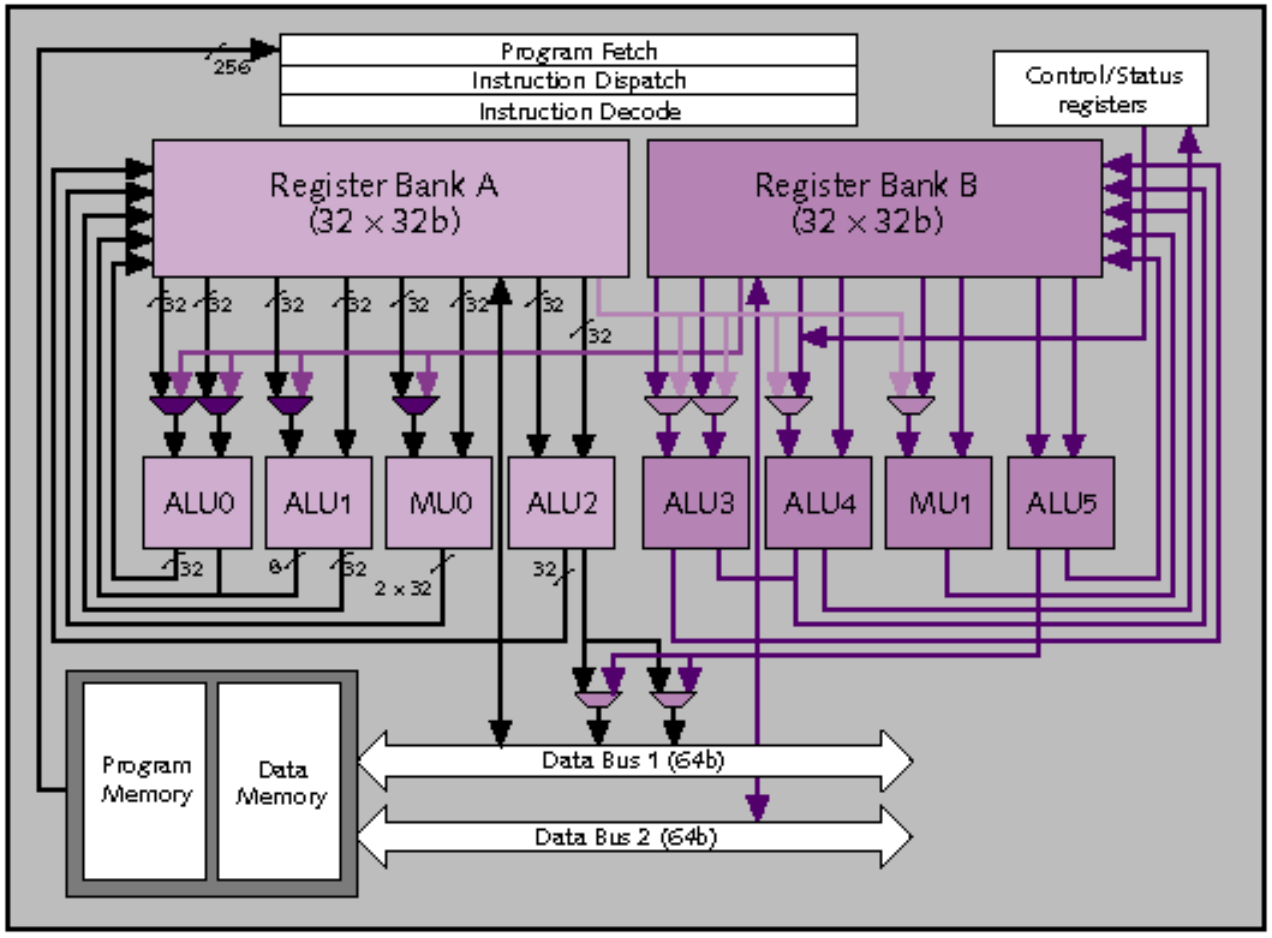
"**Fallacy**: There is a simple approach to multiple-issue processors that yields high performance without a significant investment in silicon area or design complexity"

Hennessy & Patterson, CA:AQA

VLIW Can Be Good for Embedded Processors

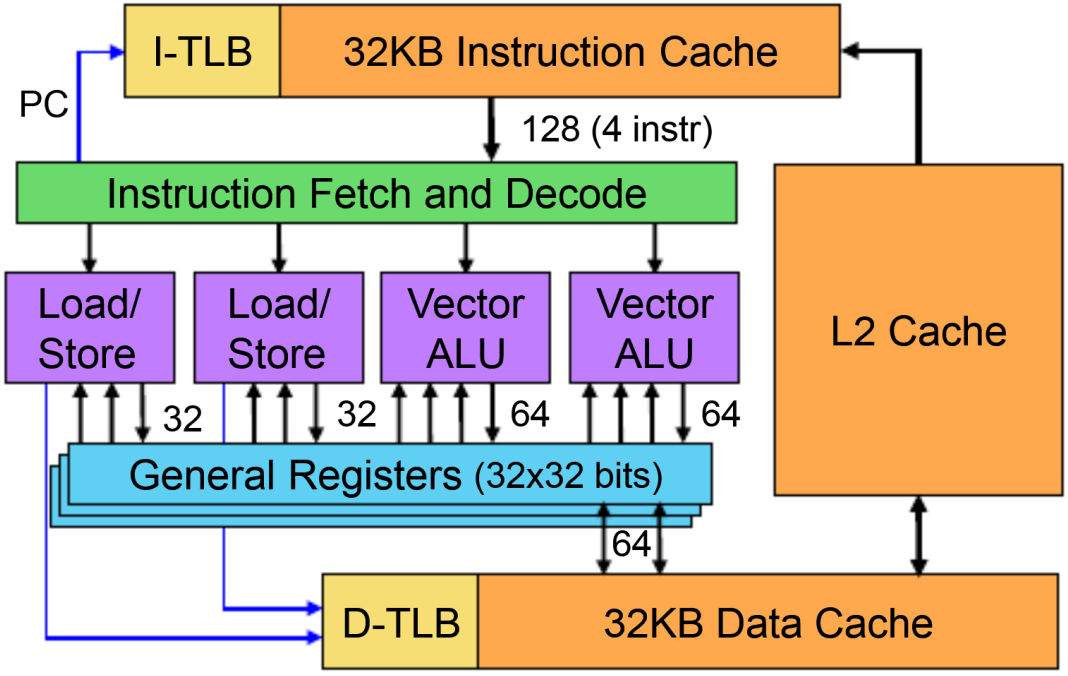
- Cost used to be the only concern; now **performance/cost is at premium** and still not performance alone as in PCs (Intel model); performance is often a constraint
- **Binary compatibility** is less of an issue for embedded systems
- Many embedded applications have an **obvious parallelism**
- **Manual** optimizations are possible (tune compiler switches, annotate code with pragmas, etc.)

Dual Cluster DSPs
TI DSP TMS320C64x



Source: Microprocessor Report, © MPR 2000

Qualcomm Hexagon v5 DSP

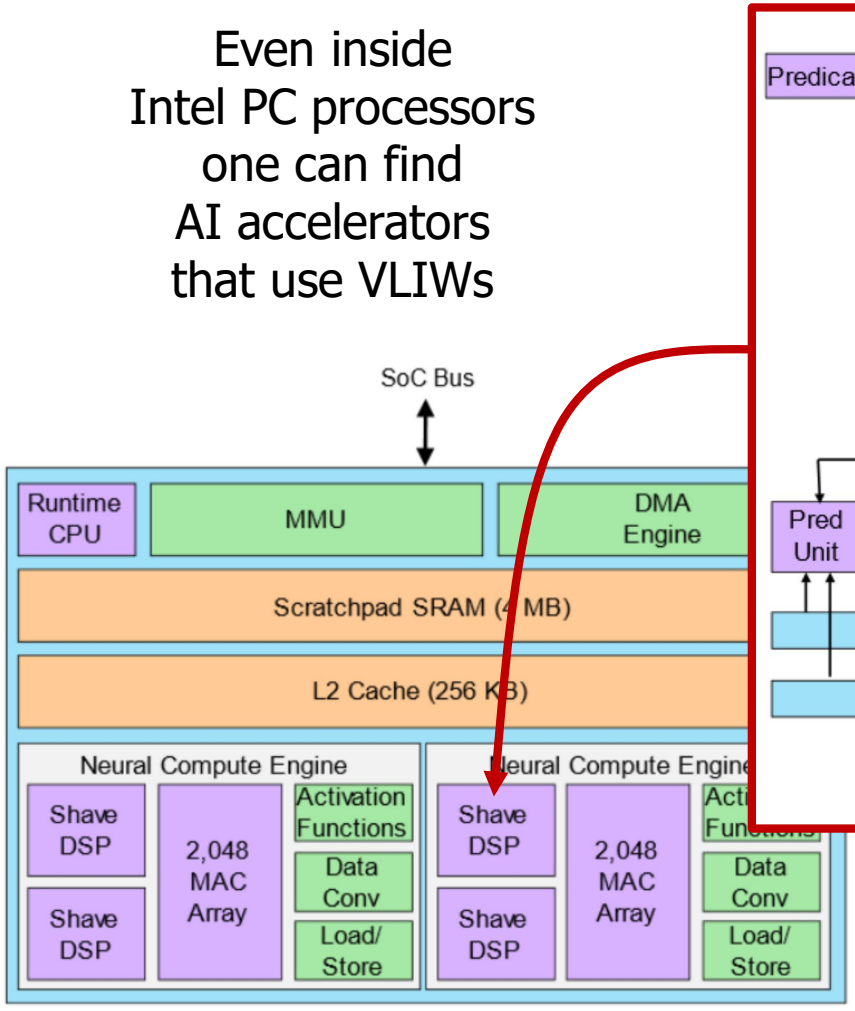


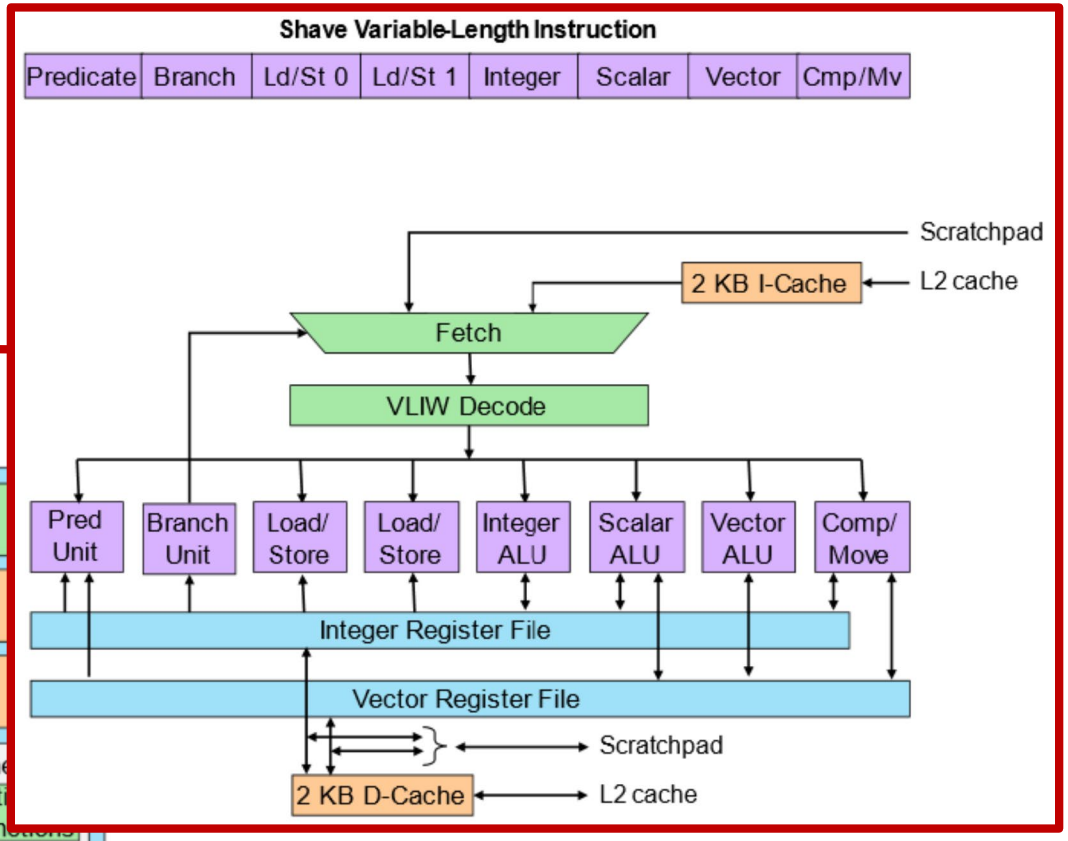
Source: Microprocessor Report, © MPR 2013

In **Qualcomm Snapdragon** since 2006

Meteor Lake AI Accelerator (NPU)

Even inside
Intel PC processors
one can find
AI accelerators
that use VLIWs





Source: Microprocessor Report, © TechInsights 2023

Google's TensorCore is a VLIW
Groq's LPU is a huge statically scheduled chip
etc.

References on VLIW

- AQA 5th ed., Appendix H
- B. R. Rau and J. A. Fisher, Instruction-Level Parallel Processing: History, Overview, and Perspective, The Journal of Supercomputing, vol. 7, p. 9-50, 1993
- M. S. Schlansker et al., *Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity*, HP Labs Technical Report HPL-96-120, November 1994

References on IA-64 and Itanium

- C. McNairy and D. Soltis, Itanium 2 Processor Microarchitecture, IEEE Micro, Mar./Apr. 2003
- S. D. Naffziger et al., *The Implementation of the Itanium 2 Microprocessor*, IEEE JSSC, November 2002
- K. Krewell, *Itanium 2 Arrives with a Benchmarking Bang*, MPR, August 2002
- J. Huck et al., *Introducing the IA-64 Architecture*, IEEE Micro, Sept./Oct. 2000
- H. Sharangpani and K. Arora, *Itanium Processor Microarchitecture*, IEEE Micro, Sept./Oct. 2000
- J. Bharadwaj, *The Intel IA-64 Compiler Code Generator*, IEEE Micro, Sept./Oct. 2000