

Advanced Computer Architecture

Chapter 6

Static instruction scheduling, for instruction-level parallelism

Software pipelining, VLIW, EPIC, instruction-set support

1	S.D	0(R1),F4	; Stores M[i]
2	ADD.D	F4,F0,F2	; Adds to M[i-1]
3	L.D	F0,-16(R1)	; Loads M[i-2]
4	DSUBUI	R1,R1,#8	
5	BNEZ	R1,LOOP	

November 2022
Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3rd and 4th eds), and on the lecture slides of David Patterson and John Kubiatowicz's Berkeley course

✦ We have seen dynamic scheduling:

➤ **out-of-order (o-o-o): exploiting instruction-level parallelism in hardware**

✦ How much of all this complexity can you shift into the compiler?

✦ What if you can also change instruction set architecture?

✦ **VLIW (Very Long Instruction Word)**

✦ **EPIC (Explicitly Parallel Instruction Computer)**

➤ Intel's (and HP's) multi-billion dollar gamble for the future of computer architecture: Itanium, IA-64

➤ Started ca.1994...not dead yet – but has it turned a profit?

Recall example from Ch02

for (i=1000; i>=0; i=i-1)
 x[i] = x[i] + s;

- **Using MIPS code:**

[For the sake of a simple example, we count *down* to location zero]

```
Loop:  L.D      F0,0(R1)  ;F0=vector element
        ADD.D   F4,F0,F2  ;add scalar from F2
        S.D     0(R1),F4  ;store result
        DSUBUI  R1,R1,8   ;decrement pointer 8B (DW)
        BNEZ    R1,Loop   ;branch R1!=zero
        NOP           ;delayed branch slot
```

Where are the stalls?

Showing Stalls

```

1 Loop: L.D      F0,0(R1) ;F0=vector element
2          stall
3          ADD.D  F4,F0,F2 ;add scalar in F2
4          stall
5          stall
6          S.D    0(R1),F4 ;store result
7          DSUBUI R1,R1,8   ;decrement pointer 8B (DW)
8          BNEZ   R1,Loop   ;branch R1!=zero
9          stall           ;delayed branch slot

```

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

 **9 clocks: Rewrite code to minimize stalls?**

Revised Loop Reducing Stalls

```

1 Loop: L.D      F0, 0(R1)
2          stall
3          ADD.D  F4, F0, F2
4          DSUBUI R1, R1, 8
5          BNEZ   R1, Loop    ;delayed branch
6          S.D    8(R1), F4   ;altered when moved past DSUBUI
  
```

Swap BNEZ and S.D by changing address of S.D

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

6 clocks, but just 3 for execution, 3 for loop overhead; How make faster?

Unroll the loop four times

- Four copies of the loop body
- One copy of increment and test
- Adjust register-indirect loads using offsets

```

1  Loop: L.D      F0, 0(R1)
2          ADD.D   F4, F0, F2
3          S.D     0(R1), F4          ;drop DSUBUI & BNEZ
4          L.D     F0, -8(R1)
5          ADD.D   F4, F0, F2
6          S.D     -8(R1), F4        ;drop DSUBUI & BNEZ
7          L.D     F0, -16(R1)
8          ADD.D   F4, F0, F2
9          S.D     -16(R1), F4       ;drop DSUBUI & BNEZ
10         L.D     F0, -24(R1)
11         ADD.D   F4, F0, F2
12         S.D     -24(R1), F4
13         DSUBUI  R1, R1, #32       ;alter to 4*8
14         BNEZ    R1, LOOP
15         NOP

```

- Re-use of registers creates WAR (“anti-dependences”)
- How can we remove them?

Loop unrolling...

```

1  Loop: L.D      F0, 0(R1)
2          ADD.D   F4, F0, F2
3          S.D     0(R1), F4          ;drop DSUBUI & BNEZ
4          L.D     F6, -8(R1)
5          ADD.D   F8, F6, F2
6          S.D     -8(R1), F8        ;drop DSUBUI & BNEZ
7          L.D     F10, -16(R1)
8          ADD.D   F12, F10, F2
9          S.D     -16(R1), F12      ;drop DSUBUI & BNEZ
10         L.D     F14, -24(R1)
11         ADD.D   F16, F14, F2
12         S.D     -24(R1), F16
13         DSUBUI  R1, R1, #32        ;alter to 4*8
14         BNEZ    R1, LOOP
15         NOP

```

The original “register renaming”

Unrolled Loop That Minimizes Stalls

```

1  Loop: L.D      F0, 0(R1)
2          L.D      F6, -8(R1)
3          L.D      F10, -16(R1)
4          L.D      F14, -24(R1)
5          ADD.D    F4, F0, F2
6          ADD.D    F8, F6, F2
7          ADD.D    F12, F10, F2
8          ADD.D    F16, F14, F2
9          S.D      0(R1), F4
10         S.D      -8(R1), F8
11         S.D      -16(R1), F12
12         DSUBUI   R1, R1, #32
13         BNEZ     R1, LOOP
14         S.D      8(R1), F16 ; 8-32 = -24

```

 **What assumptions made when moved code?**

- ➡ OK to move store past DSUBUI even though changes register
- ➡ OK to move loads before stores: get right data?
- ➡ When is it safe for compiler to make such changes?

14 clock cycles, or 3.5 per iteration

How about this?

```
1  S.D      0(R1),F4      ; Stores M[i]
2  ADD.D    F4,F0,F2      ; Adds to M[i-1]
3  L.D      F0,-16(R1)    ; Loads M[i-2]
4  DSUBUI   R1,R1,#8
5  BNEZ     R1,LOOP
```

Software Pipelining Example

Before: Unrolled 3 times

```

1  L.D    F0,0(R1)
2  ADD.D  F4,F0,F2
3  S.D    0(R1),F4
4  L.D    F6,-8(R1)
5  ADD.D  F8,F6,F2
6  S.D    -8(R1),F8
7  L.D    F10,-16(R1)
8  ADD.D  F12,F10,F2
9  S.D    -16(R1),F12
10 DSUBUI  R1,R1,#24
11 BNEZ   R1,LOOP
  
```

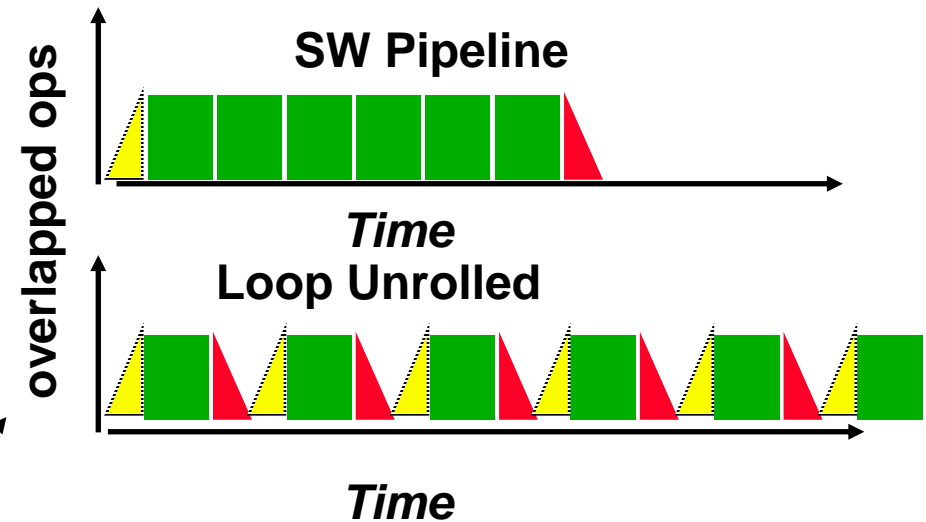
After: Software Pipelined

```

1  S.D    0(R1),F4 ; Stores M[i]
2  ADD.D  F4,F0,F2 ; Adds to M[i-1]
3  L.D    F0,-16(R1) ; Loads M[i-2]
4  DSUBUI  R1,R1,#8
5  BNEZ   R1,LOOP
  
```

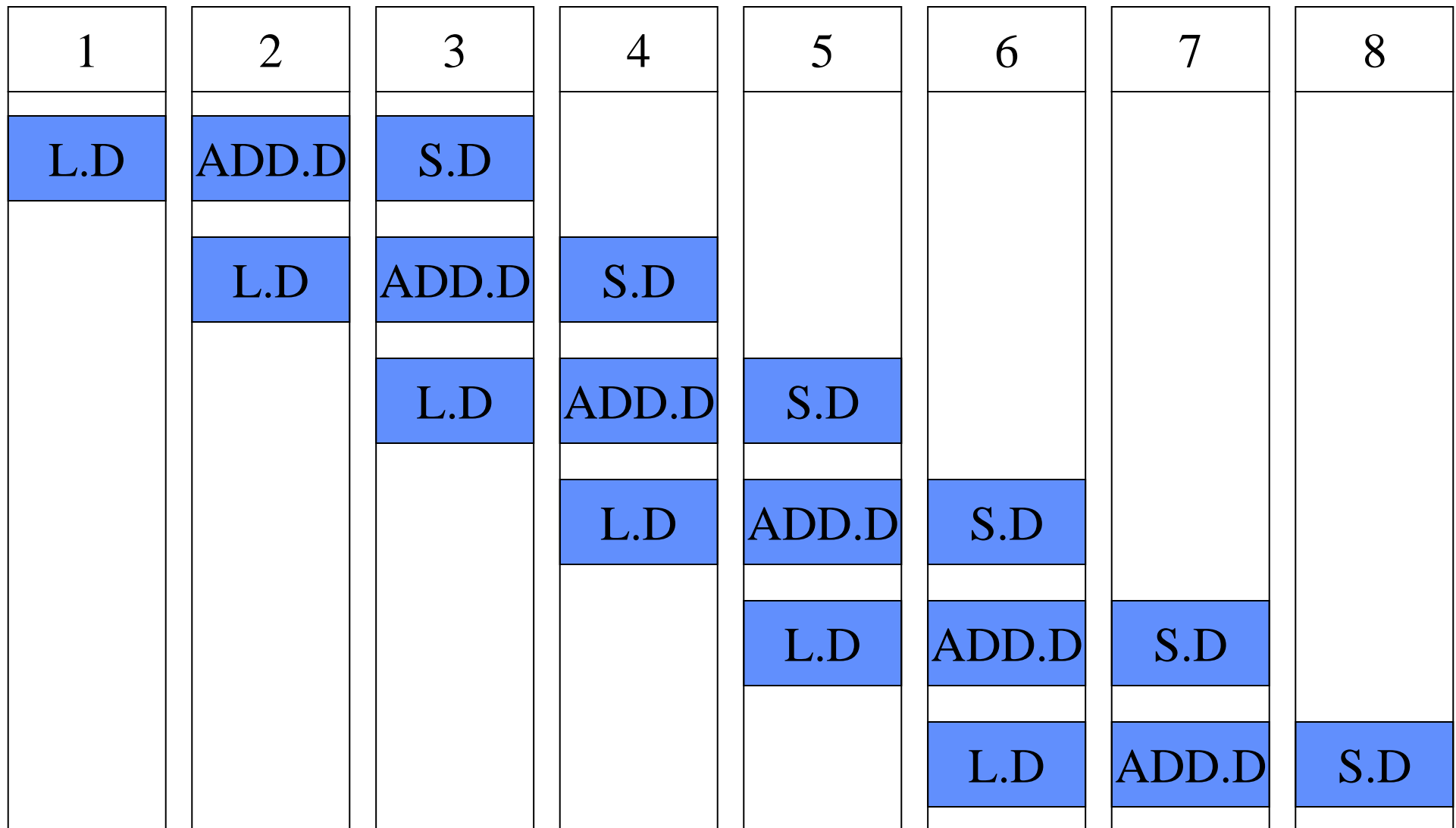
- **Symbolic Loop Unrolling**

- Maximize result-use distance
- Less code space than unrolling
- Fill & drain pipe only once per loop
vs. once per each unrolled iteration in loop unrolling



5 cycles per iteration

(3 if we can issue DSUBUI and BNEZ in parallel with other instrns)



Pipeline fills



Pipeline full



Pipeline drains

Including fill and drain phases:

Fill
phase

1	2
LD	ADD.D
	LD

```

-2 LD      F1, -0(R1) ; Loads M[N]
-1 LD      F0, -8(R1) ; Loads M[N-1]
0  ADD.D   F4, F1, F2 ; Adds to M[N]

```

```

LOOP: ; on entry, i=R1=N

```

Fully-
pipelined
phase

3	4	5	6
S.D			
ADD.D	S.D		
LD	ADD.D	S.D	
	LD	ADD.D	S.D
		LD	ADD.D
			LD

```

1  S.D      0(R1), F4 ; Stores M[i]
2  ADD.D    F4, F0, F2 ; Adds to M[i-1]
3  LD       F0, -16(R1) ; Loads M[i-2]
4  DSUBUI   R1, R1, #8
5  BNEZ     R1, LOOP

```

Drain
phase

7	8
S.D	
ADD.D	S.D

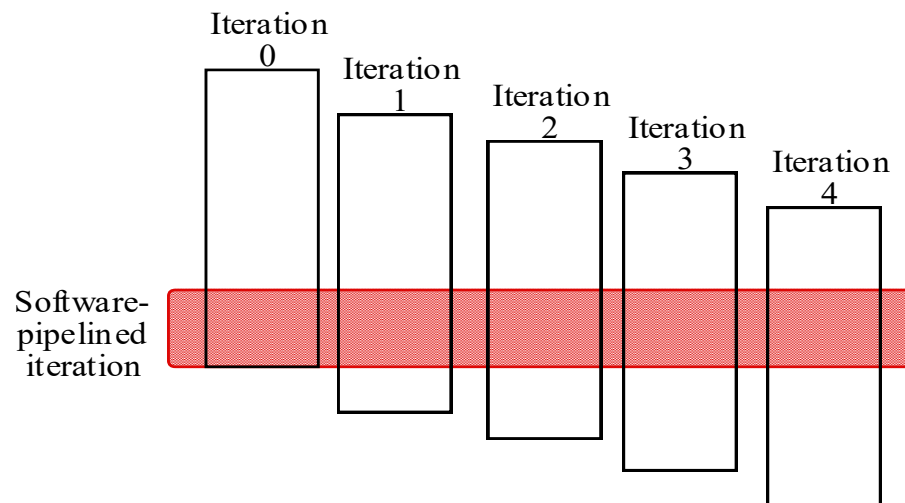
```

6  S.D      0(R1), F4 ; Stores M[i-1]
7  ADD.D    F4, F0, F2 ; Adds to M[i-2]
8  S.D      -16(R1), F4 ; Stores M[i-2]

```

Static overlapping of loop bodies: “Software Pipelining”

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (~ Tomasulo in software)



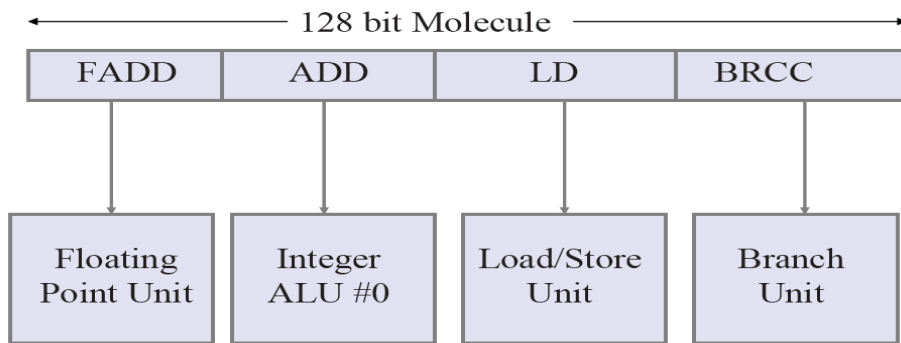
What if We Can Change the Instruction Set?

- ✦ **Superscalar processors decide on the fly how many instructions to issue in each clock cycle**
 - ➡ Have to check for dependences between all n pairs of instructions in a potential parallel issue packet
 - ➡ Hardware complexity of figuring out the number of instructions to issue is $O(n^2)$
 - Entirely doable for smallish n , but tends to lead to multiple pipeline stages between fetch and issue
- ✦ **Why not allow compiler to schedule instruction level parallelism explicitly?**
- ✦ **Format the instructions into a potential issue packet so that hardware need not check explicitly for dependences**

VLIW: Very Large Instruction Word³²

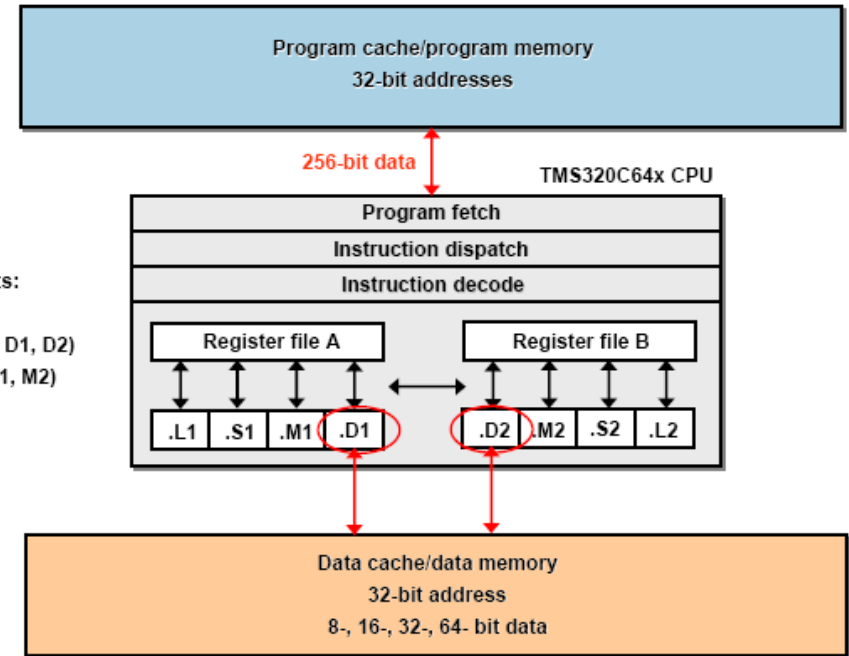
Each “instruction” has explicit coding for multiple operations

- ➡ In IA-64, grouping called a “packet”
- ➡ In Transmeta, grouping called a “molecule” (with “atoms” as ops)



Transmeta's Crusoe

Functional units:
6 ALUs
(L1, L2, S1, S2, D1, D2)
2 multipliers (M1, M2)



Texas Instruments TMS320C64x

All the operations the compiler puts in the long instruction word are independent, so can be issued and can execute in parallel

E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch

16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide

(Transmeta were cagey about details)

Need compiling technique that schedules across several branches

Recall: Unrolled Loop that Minimizes Stalls for Scalar

1	Loop:	L.D	F0, 0 (R1)	
2		L.D	F6, -8 (R1)	
3		L.D	F10, -16 (R1)	
4		L.D	F14, -24 (R1)	
5		ADD.D	F4, F0, F2	
6		ADD.D	F8, F6, F2	
7		ADD.D	F12, F10, F2	
8		ADD.D	F16, F14, F2	
9		S.D	0 (R1), F4	
10		S.D	-8 (R1), F8	
11		S.D	-16 (R1), F12	
12		DSUBUI	R1, R1, #32	
13		BNEZ	R1, LOOP	
14		S.D	8 (R1), F16	; 8-32 = -24

L.D to ADD.D: 1 Cycle
ADD.D to S.D: 2 Cycles

14 clock cycles, or 3.5 per iteration

Loop Unrolling in VLIW

<i>Memory reference 1</i>	<i>Memory reference 2</i>	<i>FP operation 1</i>	<i>FP op. 2</i>	<i>Int. op/ branch</i>	<i>Clock</i>
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				2
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		3
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
		ADD.D F20,F18,F2	ADD.D F24,F22,F2		5
S.D 0(R1),F4	S.D -8(R1),F8	ADD.D F28,F26,F2			6
S.D -16(R1),F12	S.D -24(R1),F16				7
S.D -32(R1),F20	S.D -40(R1),F24			DSUBUI R1,R1,#48	8
S.D -0(R1),F28				BNEZ R1,LOOP	9

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW (15 vs. 6)

Software Pipelining with Loop Unrolling in VLIW

<i>Memory reference 1</i>	<i>Memory reference 2</i>	<i>FP operation 1</i>	<i>FP op. 2</i>	<i>Int. op/ branch</i>	<i>Clock</i>
L.D F0,-48(R1)	ST 0(R1),F4	ADD.D F4,F0,F2			1
L.D F6,-56(R1)	ST -8(R1),F8	ADD.D F8,F6,F2		DSUBUI R1,R1,#24	2
L.D F10,-40(R1)	ST 8(R1),F12	ADD.D F12,F10,F2		BNEZ R1,LOOP	3

Software pipelined across 9 iterations of original loop

➡ In each iteration of above loop, we:

- Store to m,m-8,m-16 (iterations l-3,l-2,l-1)
- Compute for m-24,m-32,m-40 (iterations l,l+1,l+2)
- Load from m-48,m-56,m-64 (iterations l+3,l+4,l+5)

➡ 9 results in 9 cycles, or 1 clock per iteration

➡ Average: 3.67 (=11/3) instrs per clock, 73.3% utilisation (=11/15)

**Note: Need fewer registers for software pipelining
(only using 7 registers here, was using 15)**

Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

✚ IA-64: Intel’s bid to create a new instruction set architecture

- ➡ EPIC = “2nd generation VLIW”?
- ➡ ISA exposes parallelism (and many other issues) to the compiler
- ➡ But *is* binary-compatible across processor implementations

✚ Itanium™ first implementation (2001)

- ➡ 6-wide, 10-stage pipeline

✚ Itanium 2 (2002-2010)

- ➡ 6-wide, 8-stage pipeline
- ➡ <http://www.intel.com/products/server/processors/server/itanium2/>

✚ Itanium 9500 (Poulson) (2012)

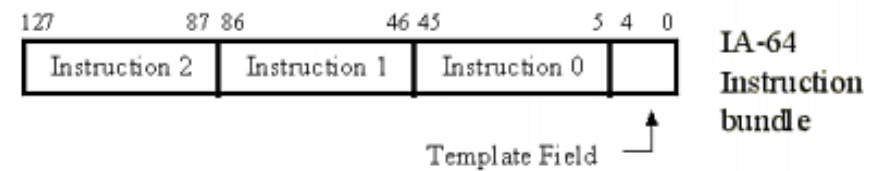
- ➡ 12-wide, 11-stage pipeline

(2017: Kittson “end of the line”)

Instruction bundling in IA-64³

➤ **Instruction group**: a sequence of consecutive instructions with no register data dependences

- All instructions in a group could be executed in parallel, if sufficient hardware resources exist and if any dependences through memory are preserved
- Instruction group can be arbitrarily long, but **compiler must explicitly indicate boundary between one instruction group and another** by placing a **stop** between 2 instructions that belong to different groups



	127				0
	Template Field				
addr N	I-unit Instr	I-unit Instr	M-unit Instr	00 ₁₆	
addr N+16	I-unit Instr	I-unit Instr	M-unit Instr	02 ₁₆	
addr N+32	I-unit Instr	I-unit Instr	M-unit Instr	03 ₁₆	
addr N+48	B-unit Instr	I-unit Instr	M-unit Instr	10 ₁₆	
addr N+64	I-unit Instr	M-unit Instr	M-unit Instr	0A ₁₆	
addr N+80	I-unit Instr	I-unit Instr	M-unit Instr	01 ₁₆	

Instruction group 1
 Instruction group 2
 Instruction group 3
 Instruction group 4
 Instruction group 5

➤ IA-64 instructions are encoded in bundles, which are 128 bits wide.

➤ Each bundle consists of a 5-bit template field and 3 instructions, each 41 bits in length

➤ One purpose of the template field is to mark where instructions in the bundle are dependent or independent, and whether they can be issued in parallel with the *next* bundle

➤ Eg for Poulson, groups of up to 4 bundles can be issued in parallel

➤ Smaller code size than old VLIW, larger than x86/RISC

Instructions can be explicitly sequential:

`add r1 = r2, r3 ;;`

`sub r4 = r1, r2 ;;`

`shl r2=r4,r8`

Or not:

`add r1 = r2, r3`

`sub r4 = r11, r21`

`shl r12 = r14, r8 ;;`

The “;;” syntax sets the “stop” bit that marks the end of a sequence of bundles that can be issued in parallel

Hardware Support for Exposing More Parallelism at Compile-Time

To help trace scheduling and software pipelining, the Itanium instruction set includes several interesting mechanisms:

- Register stack
- Predicated execution
- Speculative, non-faulting Load instructions
- Rotating register frame
- Software-assisted branch prediction
- Software-assisted memory hierarchy

**Not covered in
lecture**

➡ **Job creation scheme for compiler engineers**

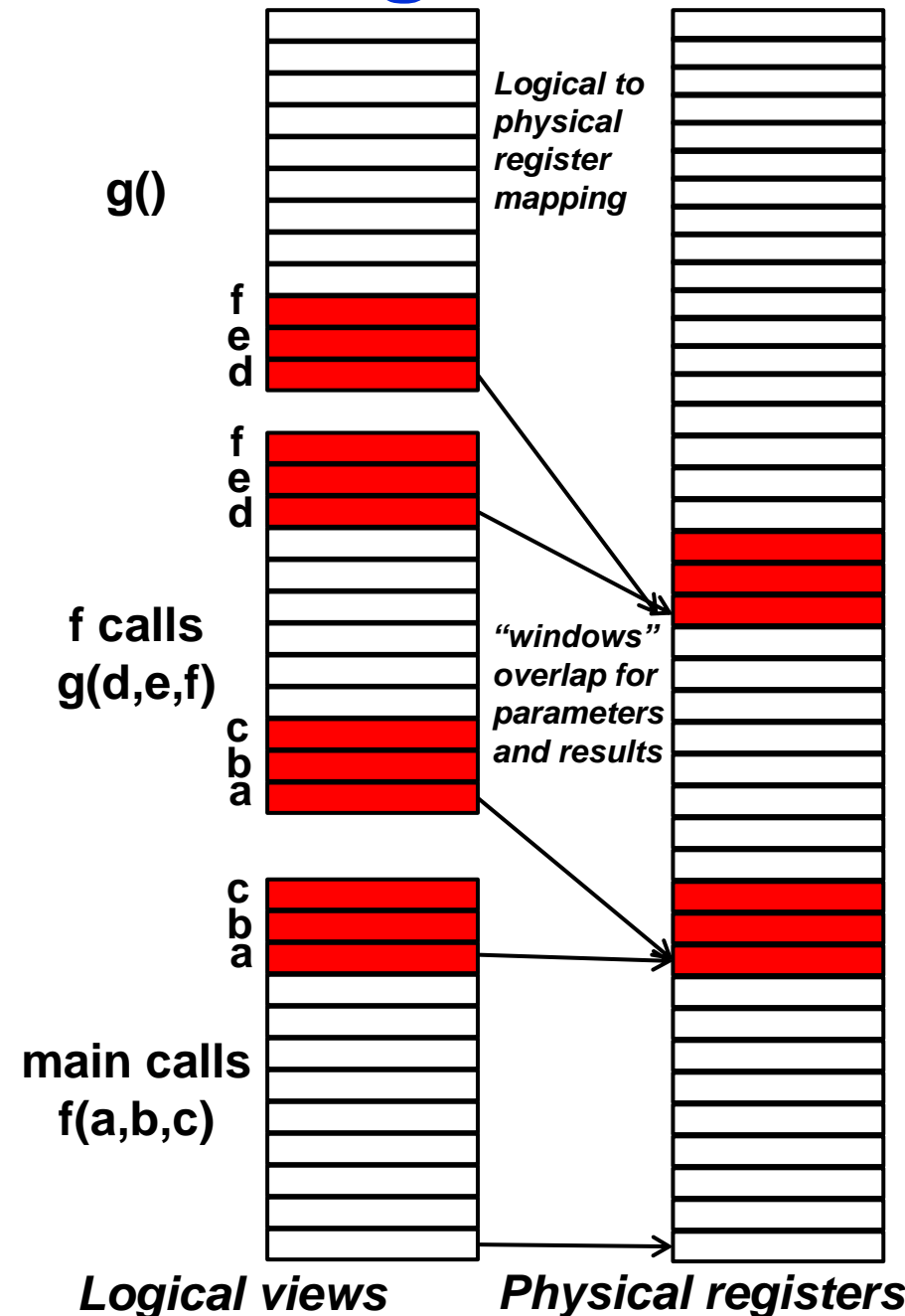
➡ **We will look at several of these in more detail**

General-purpose registers are configured to help accelerate procedure calls using a *register stack*

- Registers 32-128 are used as a register stack and each procedure is allocated a set of registers (from 0 to 96)
- The new register stack frame is created for a called procedure by renaming the registers in hardware;
- a special register called the current frame pointer (CFM) points to the set of registers to be used by a given procedure
- Registers 0-31 are always accessible and addressed as 0-31

(Mechanism similar to that developed in the Berkeley RISC-I processor and used in the SPARC architecture)

IA-64 register stack



Predication...

64 1-bit predicate registers

(p1) add r1 = r2, r3

// executed if p1

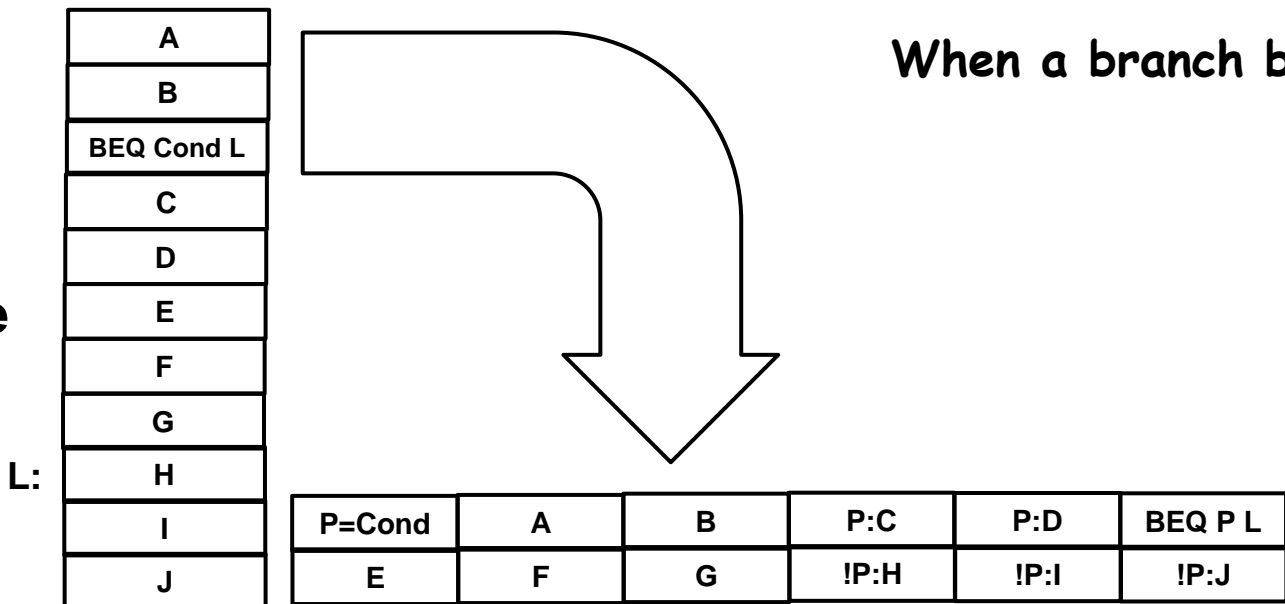
(p2) sub r1 = r2, r3 ;;

// executed if p2

shl r12 = r1, r8 J *// executed always*

Predication means

- ➡ Compiler can move instructions across conditional branches
- ➡ To pack parallel issue groups
- ➡ May also eliminate some conditional branches completely
- ➡ Avoiding branch prediction and misprediction



Predication...

64 1-bit predicate registers

(p1) add r1 = r2, r3

// executed if p1

(p2) sub r1 = r2, r3 ;;

// executed if p2

shl r12 = r1, r8

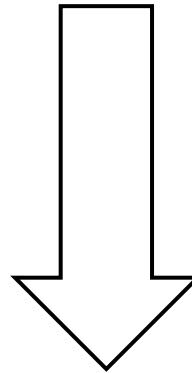
// executed always

Predication means

- ➡ Compiler can move instructions across conditional branches
- ➡ To pack parallel issue groups
- ➡ May also eliminate some conditional branches completely
- ➡ Avoiding branch prediction and misprediction

L:

A	B	BEQ Cond L		
C	D	E	F	G
H	I	J		



When a branch would break a parallel issue packet, move instructions and predicate them

L:

P=Cond	A	B	P:C	P:D	BEQ P L
E	F	G	!P:H	!P:I	!P:J

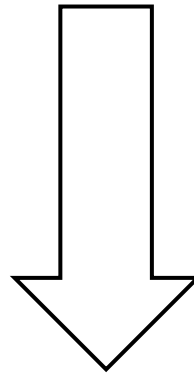
Predication...

Predication means

- ➡ Compiler can move instructions across conditional branches
- ➡ To pack parallel issue groups
- ➡ May also eliminate some conditional branches completely
- ➡ Avoiding branch prediction and misprediction

Parallel issue packets

	BEQ Cond L	Lost due to branch				
	A	B	C	D	E	F
L:	G	H	Lost due to label			
	I	J	K	L	L	L



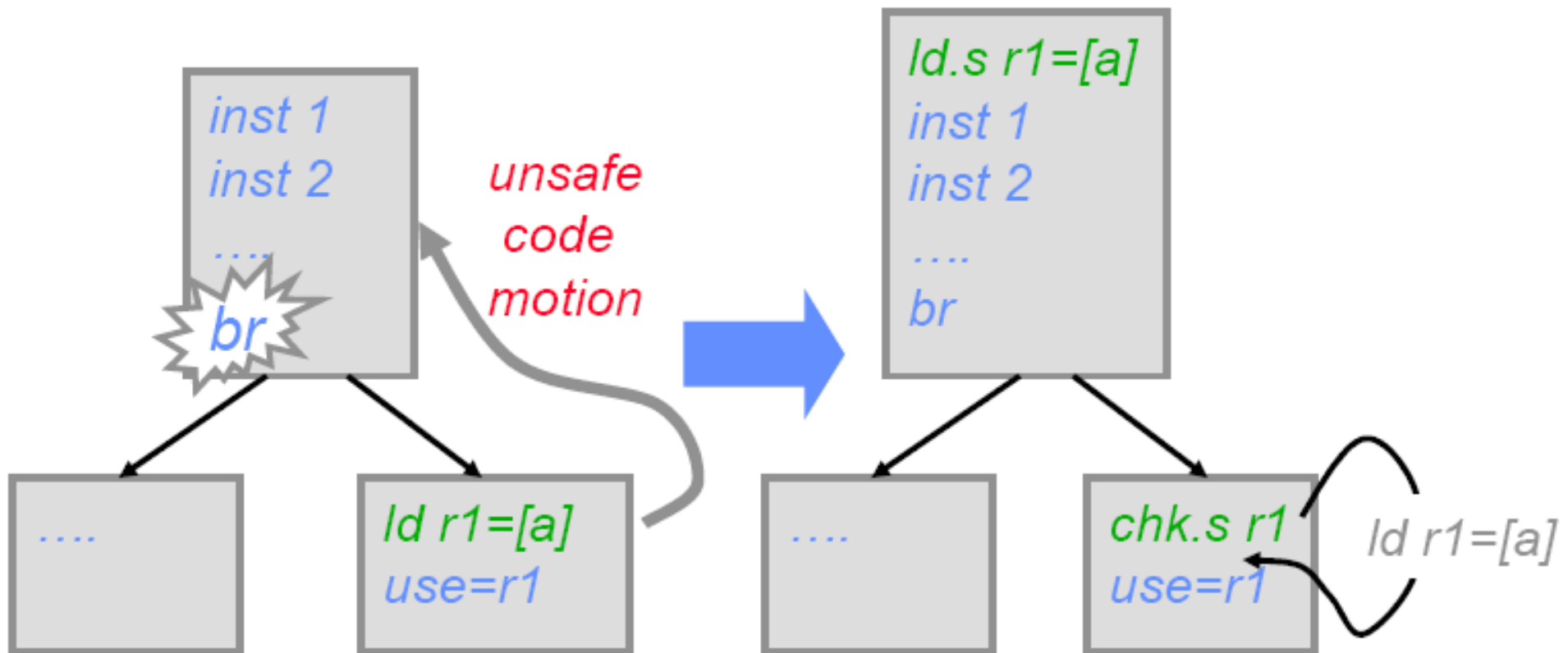
When a branch would break a parallel issue packet, move instructions and predicate them

	P=Cond	(P) A	(P) B	(P) C	(P) D	BEQ P L
	E	F				
L:	(!P) G	(!P) H	I	J	K	L

IA64 load instruction variants

- IA64 has several different mechanisms to enable the compiler to schedule loads
- ld.s** – speculative, non-faulting
- ld.a** – speculative, “advanced” – checks for aliasing stores
- Register values may be marked “NaT” – not a thing
 - ➡ If speculation was invalid
- Advanced Load Address Table (ALAT) tracks stores to addresses of “advanced” loads

IA64: Speculative, Non-Faulting Load

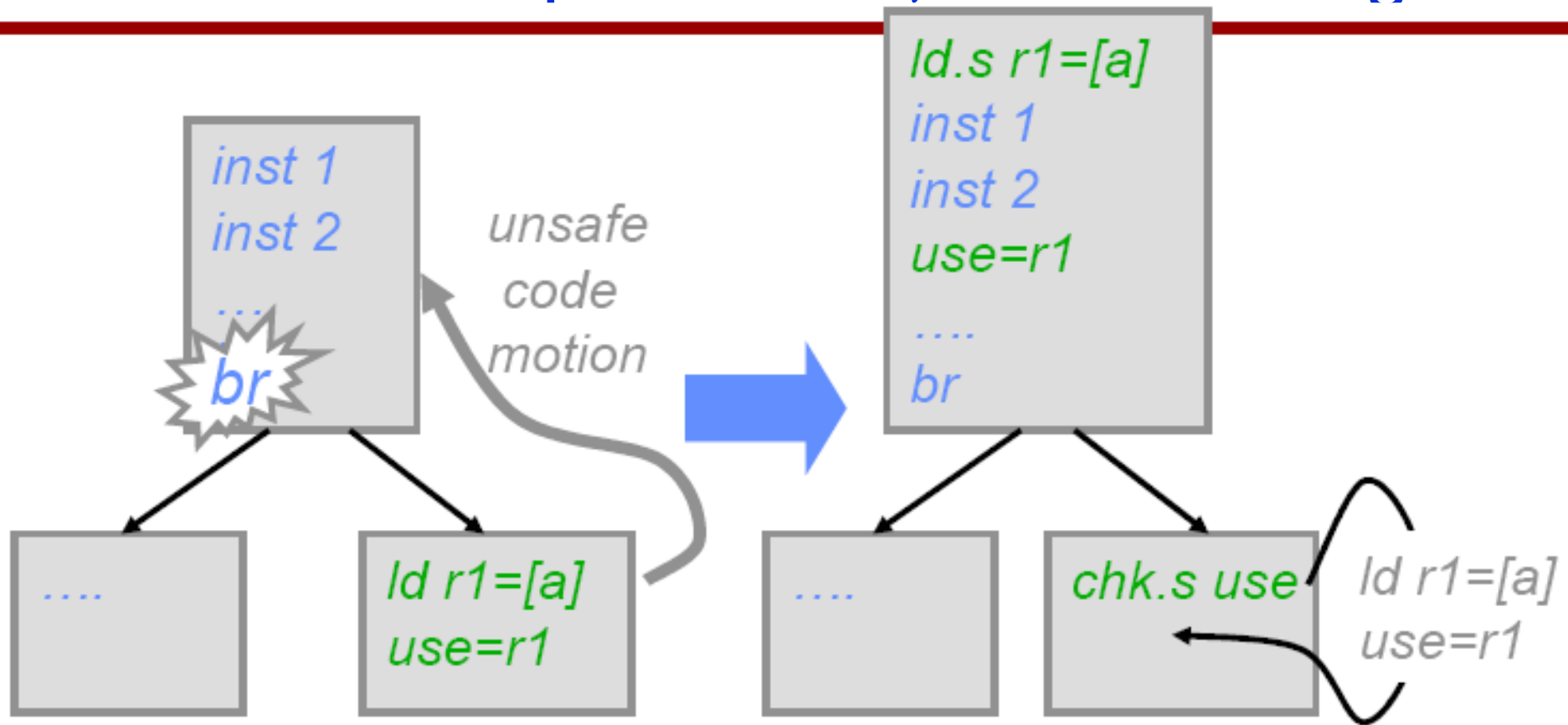


ld.s fetches speculatively from memory

► i.e. any exception due to **ld.s** is suppressed

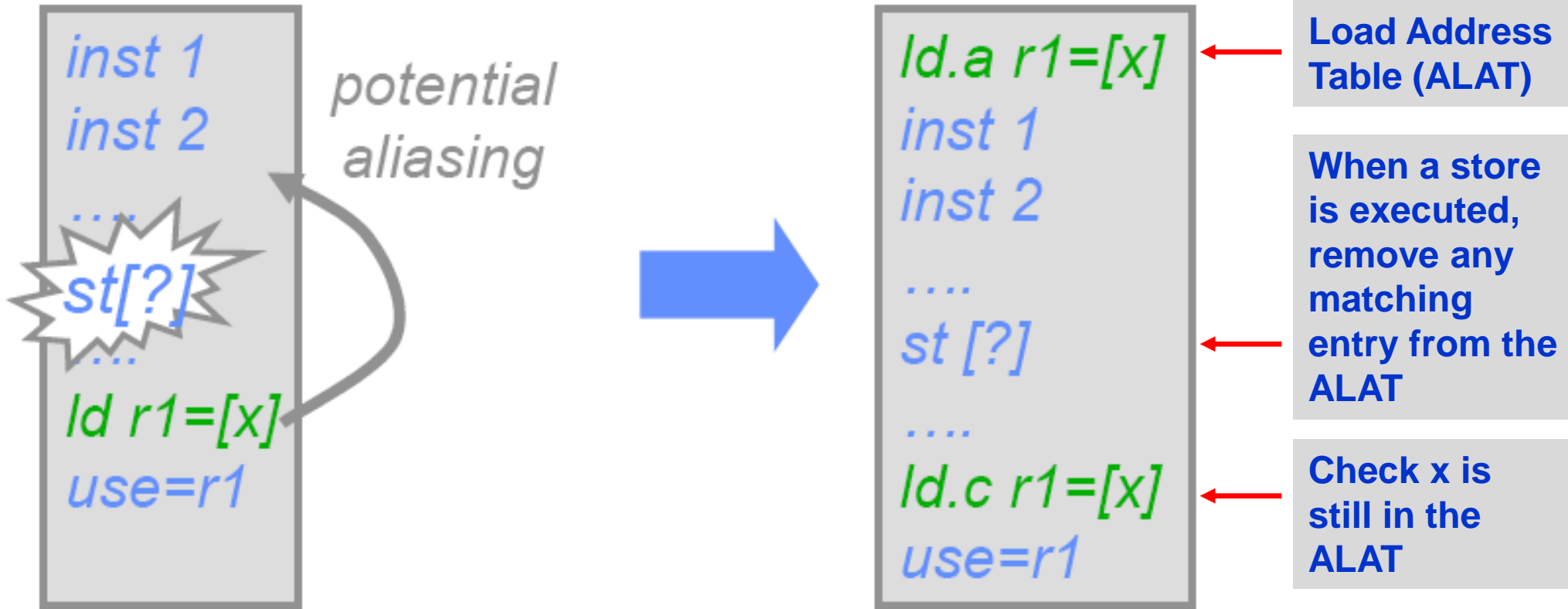
ld.s r did not cause an exception then **chk.s r** is an NOP, else a branch is taken to some compensation code

IA64: Speculative, Non-Faulting Load



- Speculatively-loaded data can be consumed prior to check
- “speculation” status is propagated with speculated data via NaT
- Any instruction that uses a speculative result also becomes speculative
 - ➡ (i.e. suppressed exceptions)
- chk.s** checks the entire dataflow sequence for exceptions

IA64: Speculative “Advanced” Load



- **ld.a** starts the monitoring of any store to the same address as the advanced load
- If no aliasing has occurred since **ld.a**, **ld.c** is a NOP
- If aliasing has occurred, **ld.c** re-loads from memory

IA-64 Registers

- Both the integer and floating point registers support register rotation for registers 32-128.
- Register rotation** is designed to ease the task of register allocation in software pipelined loops
- When combined with predication, possible to avoid the need for unrolling and for separate prologue and epilogue code for a software pipelined loop
 - makes software pipelining usable for loops with smaller numbers of iterations, where the overheads would traditionally negate many of the advantages

How Register Rotation Helps Software Pipelining

Consider this loop for copying data:

```
L1:    ld4    r35 = [r4], 4    // post-increment by 4
       st4    [r5] = r37, 4    // post-increment by 4
       br.ctop L1 ;;
```



The br.ctop instruction in the example rotates the general registers (actually br.ctop does more as we shall see)

Therefore the value stored into r35 is read in r37 two iterations (and two rotations) later.

The register rotation eliminated a dependence between the load and the store instructions, and allowed the loop to execute in one cycle.

 The logical-to-physical register mapping is shifted by 1 each time the branch ("br.ctop") is executed

Software Pipelining Example in the IA-64

```
mov pr.rot      = 0      // Clear all rotating predicate registers
cmp.eq p16,p0 = r0,r0    // Set p16=1
mov ar.lc       = 4      // Set loop counter to n-1
mov ar.ec       = 3      // Set epilog counter to 3
```

...

loop:

```
(p16) ldl r32 = [r12], 1    // Stage 1: load x
(p17) add r34 = 1, r33      // Stage 2: y=x+1
(p18) stl [r13] = r35,1    // Stage 3: store y
      br.ctop loop         // Branch back
```



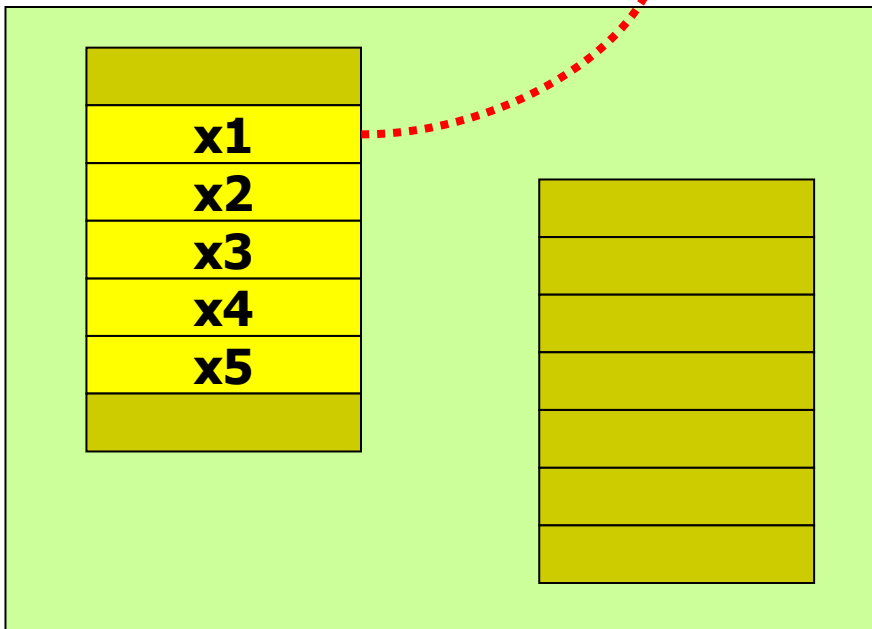
One issue packet

- Predicate mechanism activates successive stages of the software pipeline, to fill on start-up and drain when the loop terminates
- The software pipeline branch “br.ctop” rotates the predicate registers, and injects a 1 into p16
- Thus enabling one stage at a time, for execution of prologue
- When loop trip count is reached, “br.ctop” injects 0 into p16, disabling one stage at a time, then finally falls-through

Software Pipelining Example in the IA-64

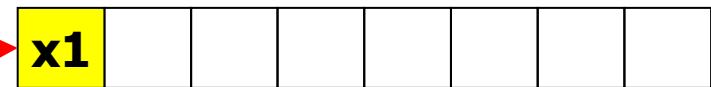
loop:
 (p16) **ldl r32 = [r12], 1**
 (p17) **add r34 = 1, r33**
 (p18) **stl [r13] = r35, 1**
br.ctop loop

Memory



General Registers (Physical)

32 33 34 35 36 37 38 39



32 33 34 35 36 37 38 39

General Registers (Logical)

Predicate Registers



16 17 18

LC



EC



RRB

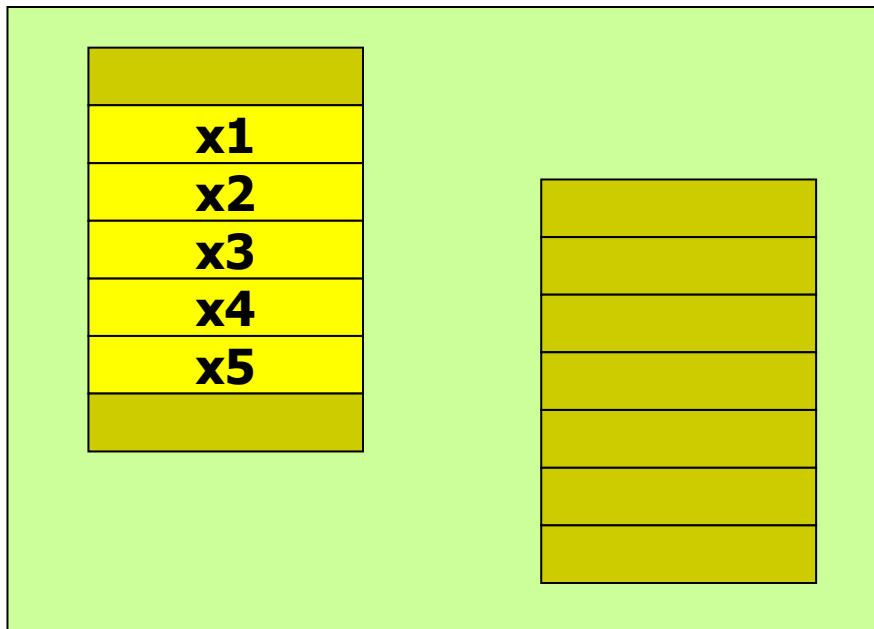


Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32  = [r12], 1
(p17) add r34  = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
  
```

Memory



General Registers (Physical)

32 33 34 35 36 37 38 39

x1							
----	--	--	--	--	--	--	--

32 33 34 35 36 37 38 39

General Registers (Logical)

Predicate Registers

1	0	0
---	---	---

16 17 18

LC

4

EC

3

RRB

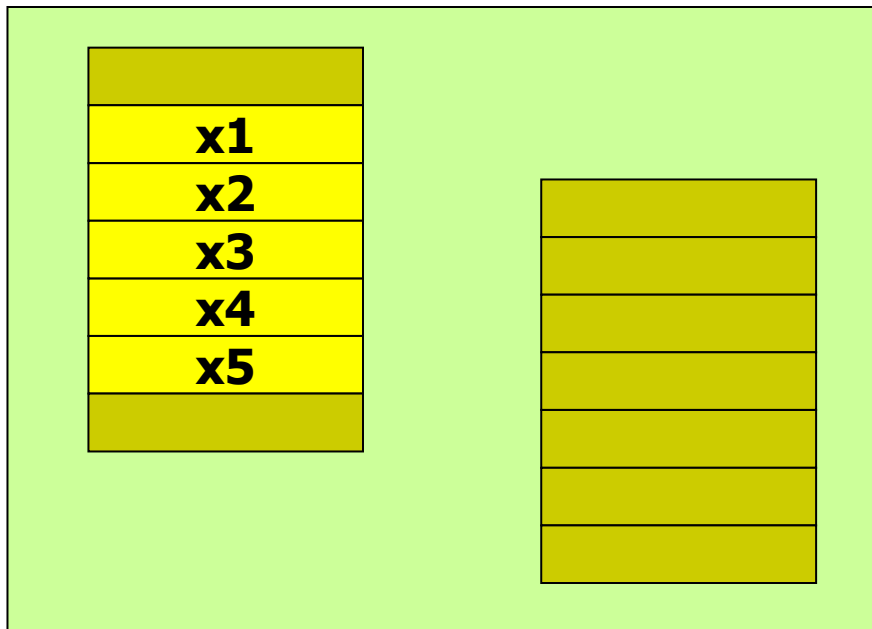
0

Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32  = [r12], 1
(p17) add r34  = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
  
```

Memory



General Registers (Physical)

32 33 34 35 36 37 38 39

x1							
----	--	--	--	--	--	--	--

32 33 34 35 36 37 38 39
General Registers (Logical)

Predicate Registers

1	0	0
---	---	---

16 17 18

LC

4

EC

3

RRB

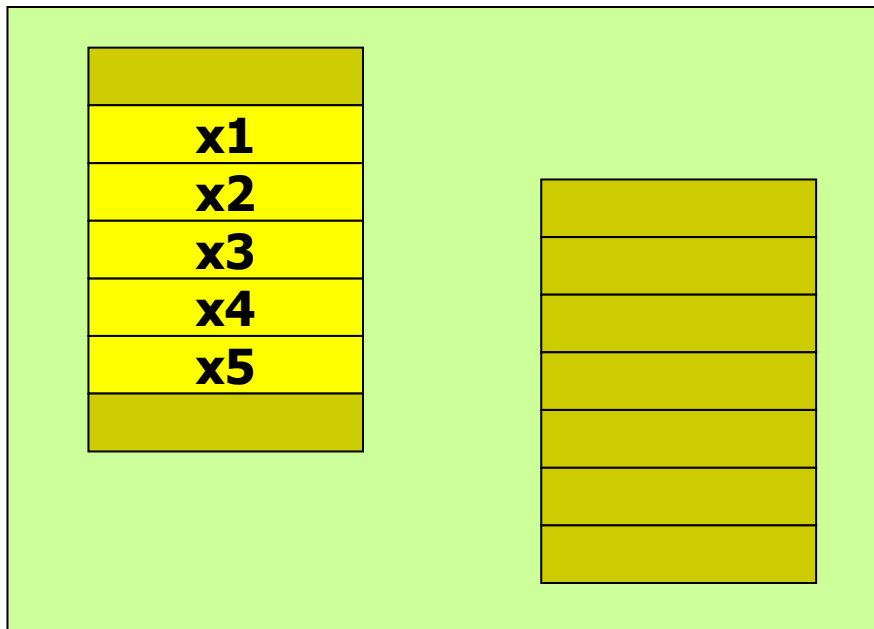
0

Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32  = [r12], 1
(p17) add r34  = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
  
```

Memory



General Registers (Physical)

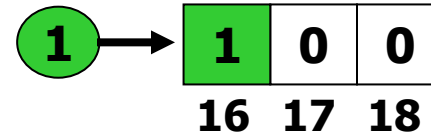
32 33 34 35 36 37 38 39

x1							
----	--	--	--	--	--	--	--

33 34 35 36 37 38 39 32

General Registers (Logical)

Predicate Registers



LC

4

EC

3

RRB

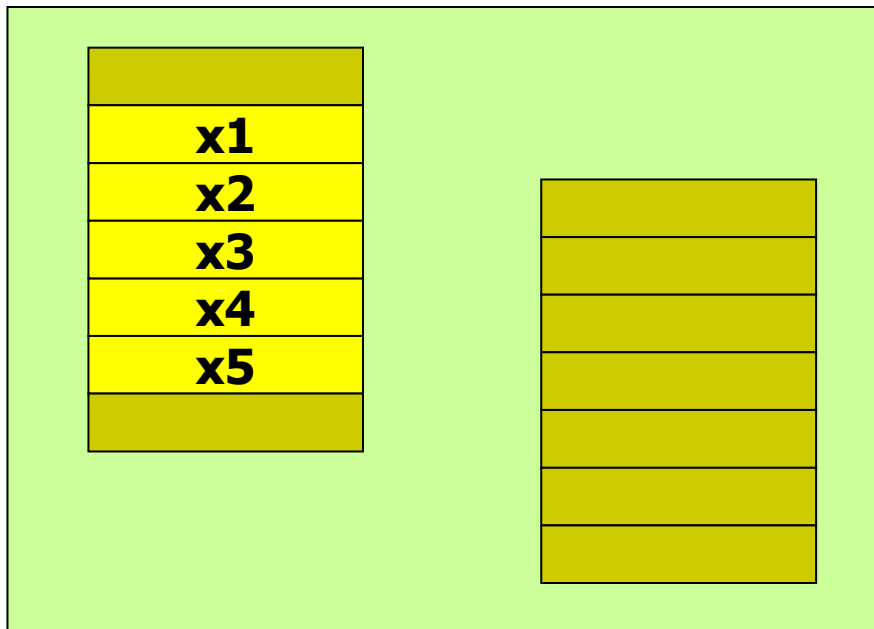
-1

Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32  = [r12], 1
(p17) add r34  = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
  
```

Memory



General Registers (Physical)

32 33 34 35 36 37 38 39

x1							
----	--	--	--	--	--	--	--

33 34 35 36 37 38 39 32

General Registers (Logical)

Predicate Registers

1	1	0
---	---	---

16 17 18

LC

3

EC

3

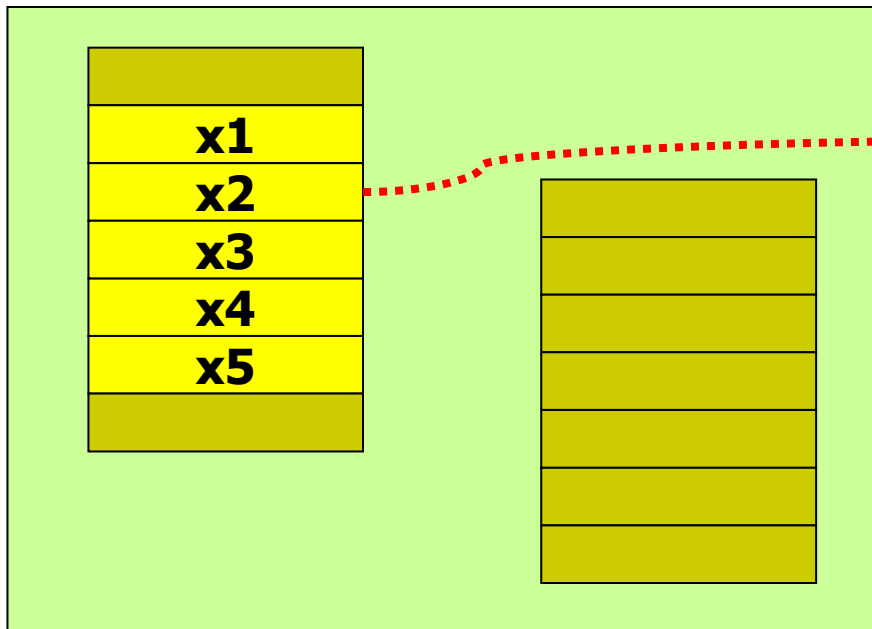
RRB

-1

Software Pipelining Example in the IA-64

loop:
 (p16) **ldl r32 = [r12], 1**
 (p17) **add r34 = 1, r33**
 (p18) **stl [r13] = r35, 1**
br.ctop loop

Memory



General Registers (Physical)

32 33 34 35 36 37 38 39



General Registers (Logical)

33 34 35 36 37 38 39 32

Predicate Registers



16 17 18

LC

3

EC

3

RRB

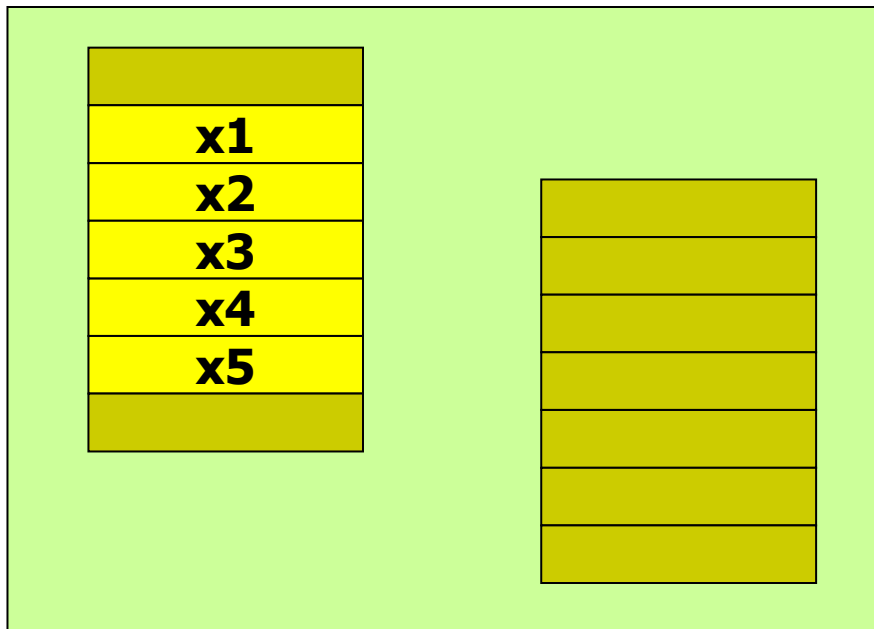
-1

Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32  = [r12], 1
(p17) add r34  = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
  
```

Memory



General Registers (Physical)

32 33 34 35 36 37 38 39



33 34 35 36 37 38 39 32
General Registers (Logical)

Predicate Registers



16 17 18

LC



EC



RRB

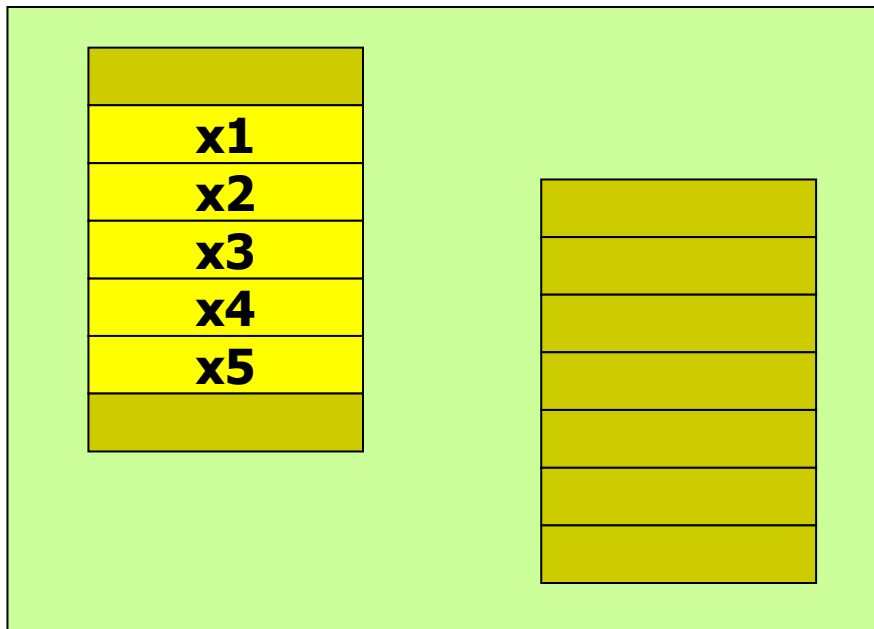


Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32  = [r12], 1
(p17) add r34  = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
  
```

Memory



General Registers (Physical)

32 33 34 35 36 37 38 39



General Registers (Logical)

33 34 35 36 37 38 39 32

Predicate Registers



16 17 18

LC



EC



RRB

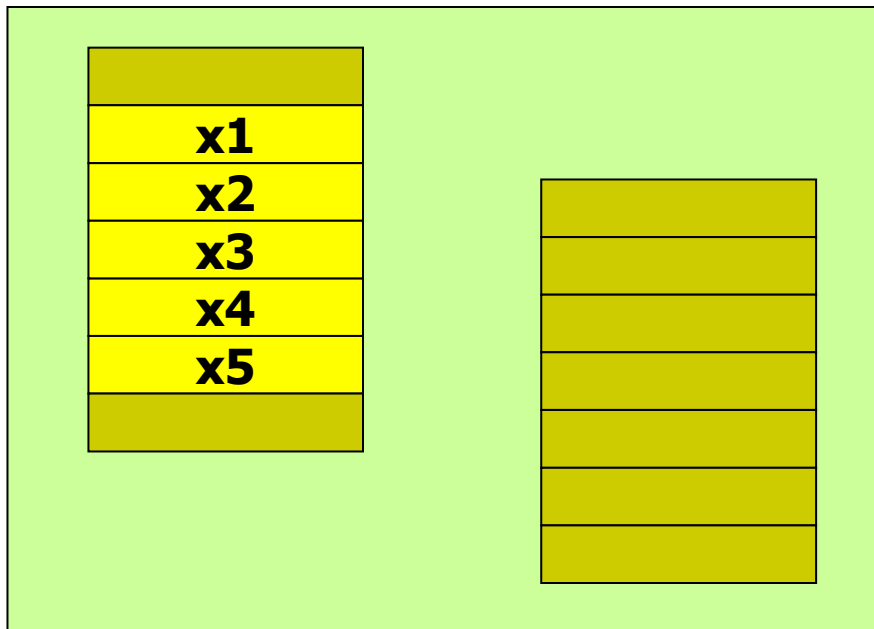


Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32  = [r12], 1
(p17) add r34  = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
  
```

Memory



General Registers (Physical)

32 33 34 35 36 37 38 39



General Registers (Logical)

33 34 35 36 37 38 39 32

Predicate Registers



16 17 18

LC



EC



RRB

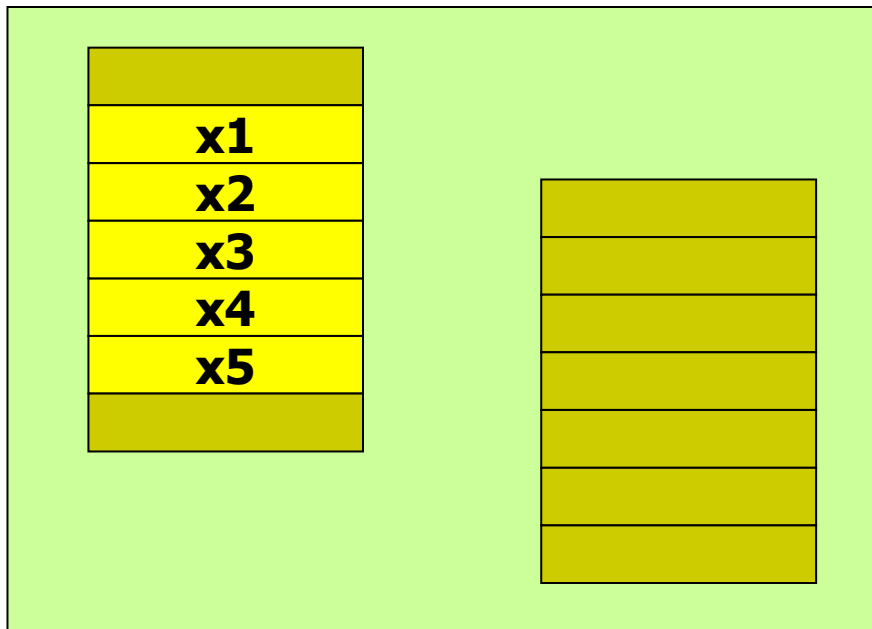


Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32  = [r12], 1
(p17) add r34  = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
  
```

Memory



General Registers (Physical)

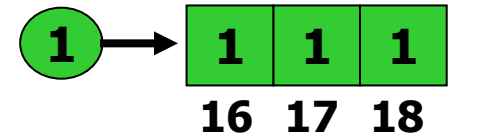
32 33 34 35 36 37 38 39



General Registers (Logical)

34 35 36 37 38 39 32 33

Predicate Registers



LC



EC



RRB

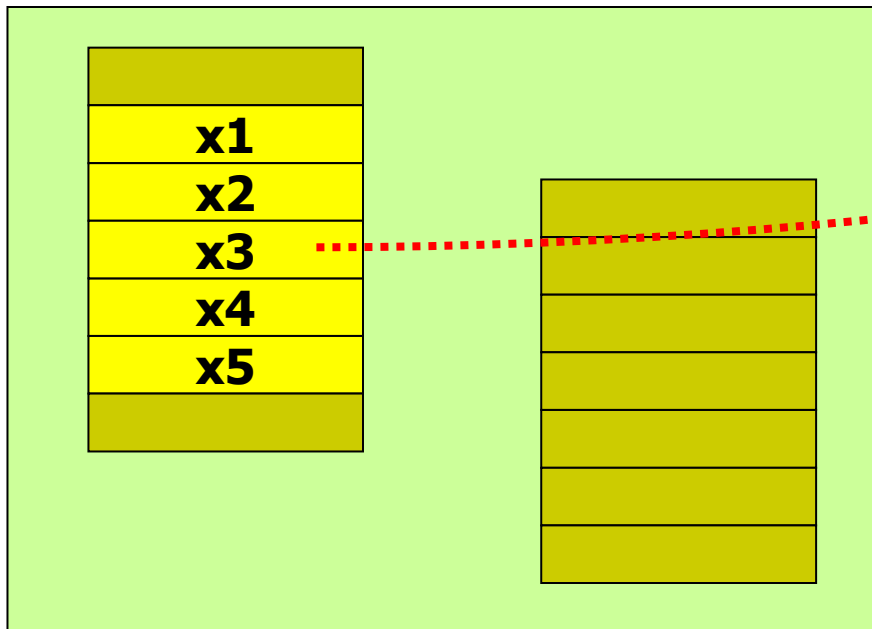


Software Pipelining Example in the IA-64

```

loop:
(p16)  ldl r32  = [r12], 1
(p17)  add r34  = 1, r33
(p18)  stl [r13] = r35,1
      br.ctop loop
  
```

Memory



General Registers (Physical)

32 33 34 35 36 37 38 39



General Registers (Logical)

34 35 36 37 38 39 32 33

Predicate Registers



16 17 18

LC



EC



RRB

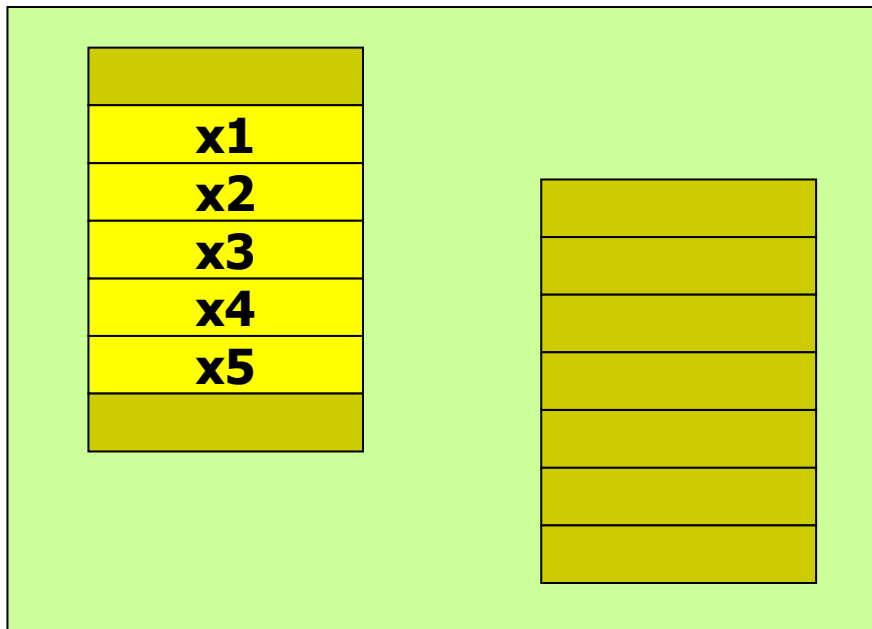


Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32  = [r12], 1
(p17) add r34  = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
  
```

Memory



General Registers (Physical)

32	33	34	35	36	37	38	39
y2	y1					x3	x2

General Registers (Logical)

Predicate Registers

1	1	1
---	---	---

16 17 18

LC

2

EC

3

RRB

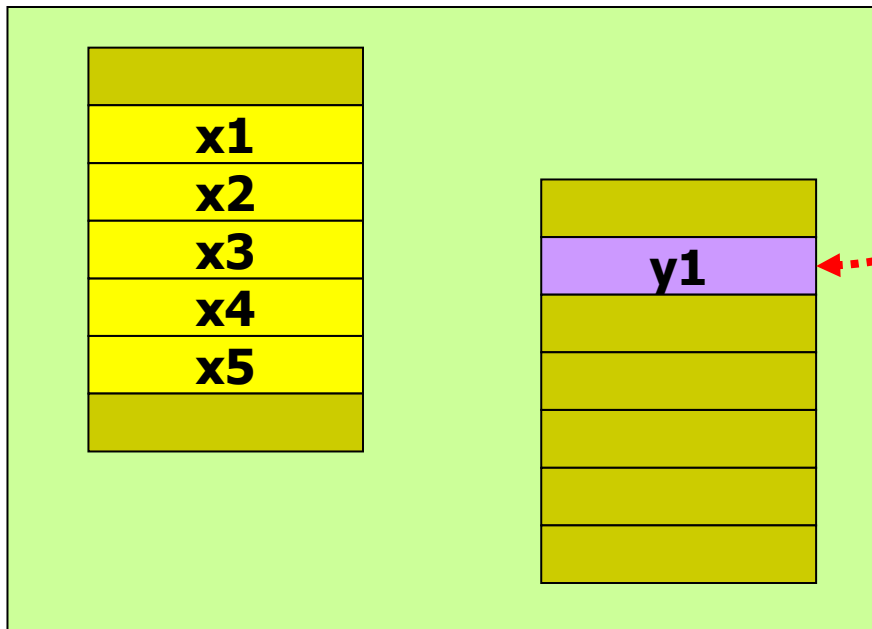
-2

Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32  = [r12], 1
(p17) add r34  = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
  
```

Memory



General Registers (Physical)

32	33	34	35	36	37	38	39
y2	y1					x3	x2

General Registers (Logical)

Predicate Registers

1	1	1
---	---	---

16 17 18

LC

2

EC

3

RRB

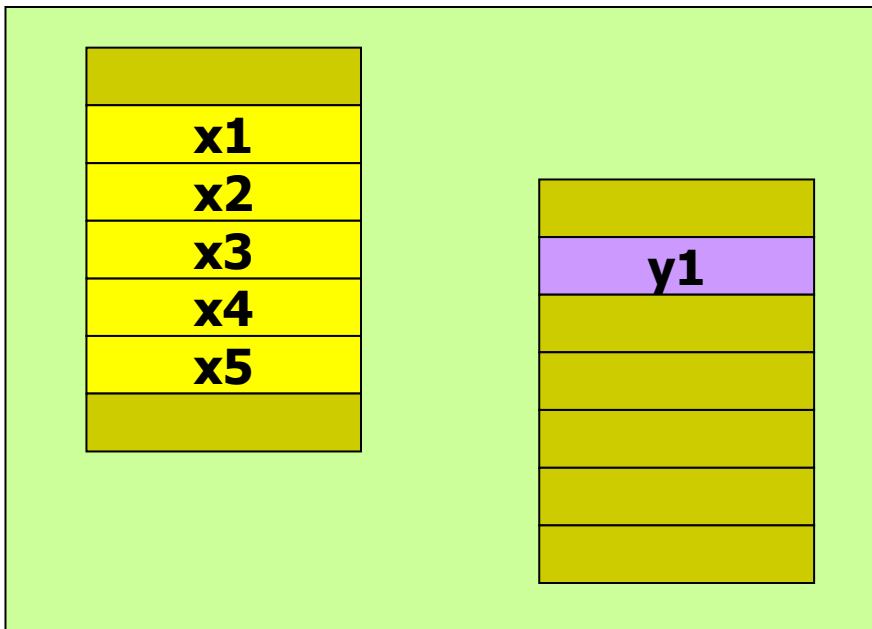
-2

Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32  = [r12], 1
(p17) add r34  = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
  
```

Memory



General Registers (Physical)

32	33	34	35	36	37	38	39
y2	y1					x3	x2

General Registers (Logical)

Predicate Registers

1	1	1
---	---	---

16 17 18

LC

2

EC

3

RRB

-2

Execution continues...

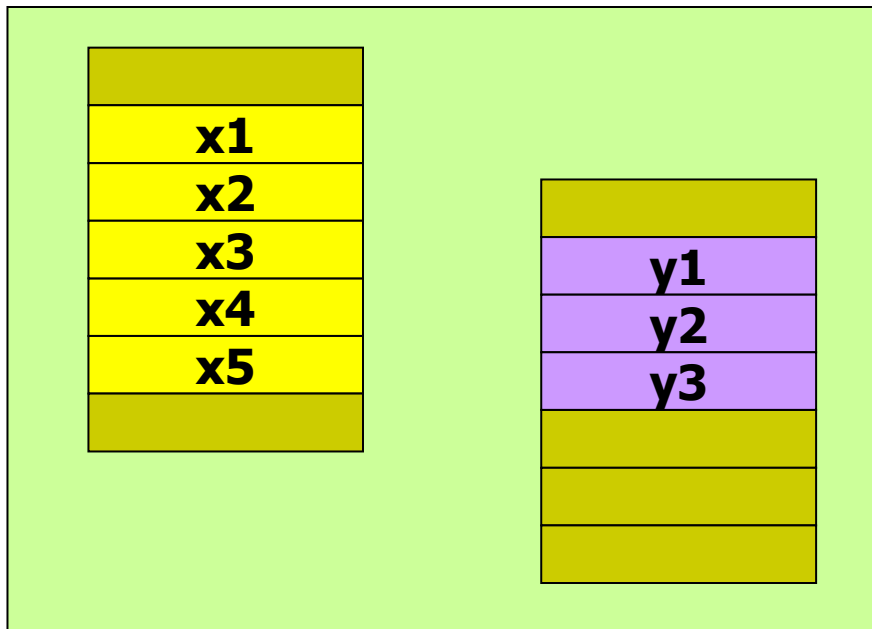
- ▶ In the central phase all stages of the software pipeline are active – all predicate bits are set
- ▶ We continue with start of pipeline drain phase

Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32  = [r12], 1
(p17) add r34  = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
  
```

Memory



General Registers (Physical)

32	33	34	35	36	37	38	39
y2	y1			x5	x4	y4	y3

General Registers (Logical)

Predicate Registers

1	1	1
---	---	---

16 17 18

LC

0

EC

3

RRB

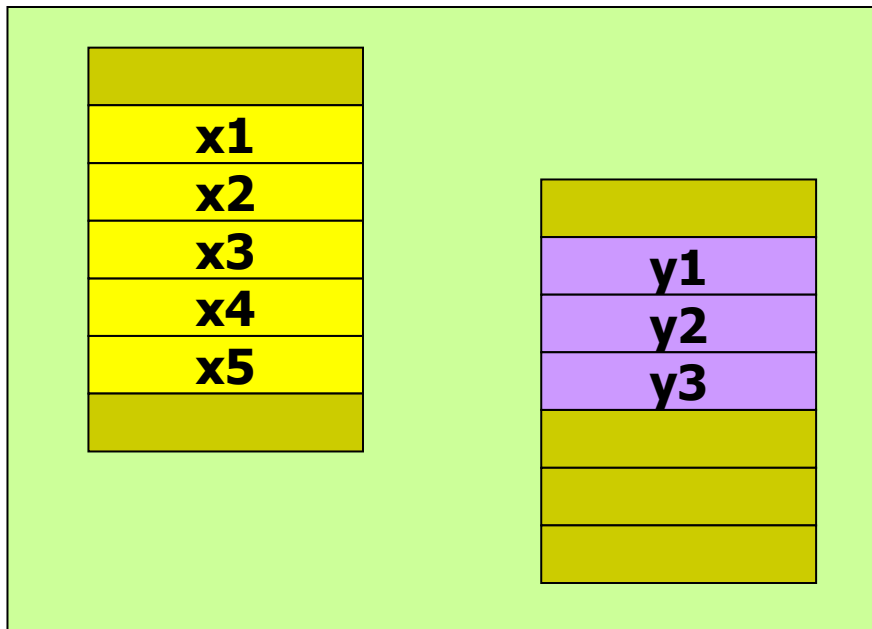
-4

Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32  = [r12], 1
(p17) add r34  = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
  
```

Memory

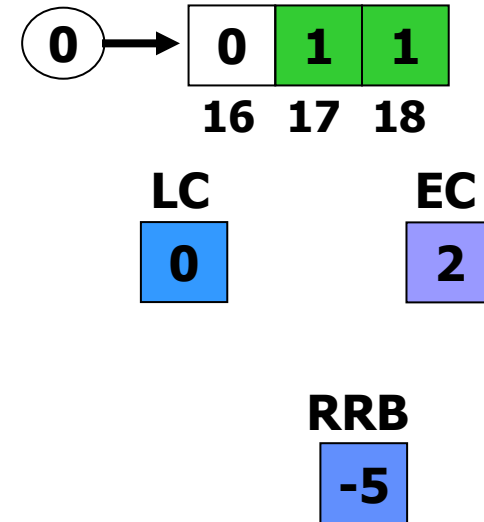


General Registers (Physical)

32	33	34	35	36	37	38	39
y2	y1			x5	x4	y4	y3

General Registers (Logical)

Predicate Registers

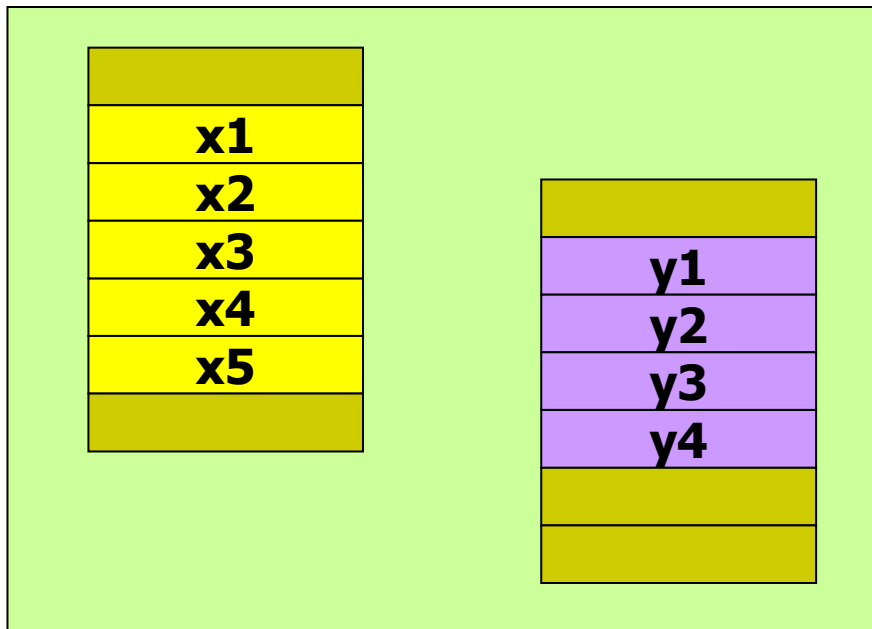


Software Pipelining Example in the IA-64

```

loop:
(p16)  ldl r32  = [r12], 1
(p17)  add r34  = 1, r33
(p18)  stl [r13] = r35,1
      br.ctop loop
  
```

Memory

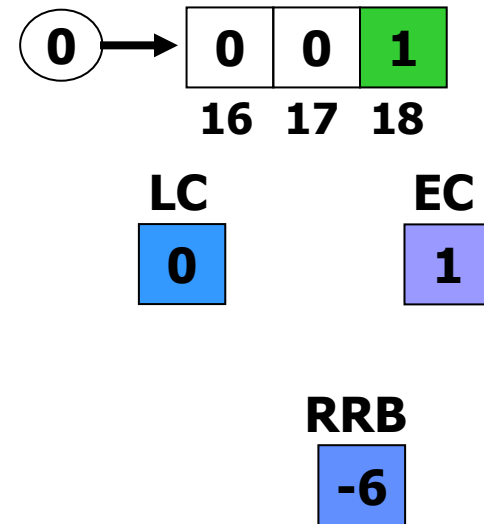


General Registers (Physical)

32	33	34	35	36	37	38	39
y2	y1			x5	y5	y4	y3

General Registers (Logical)

Predicate Registers

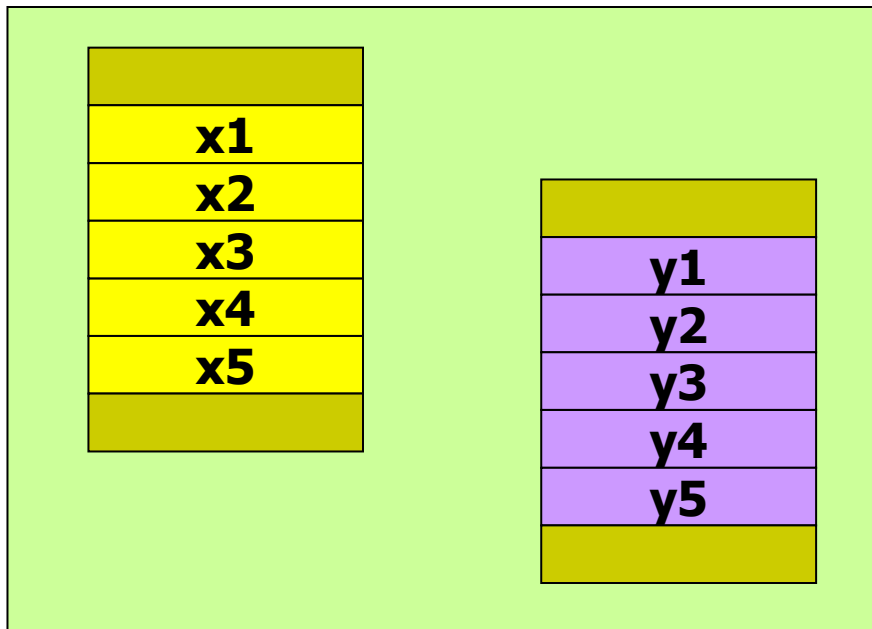


Software Pipelining Example in the IA-64

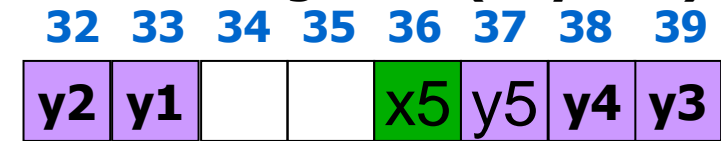
```

loop:
(p16)  ldl r32  = [r12], 1
(p17)  add r34  = 1, r33
(p18)  stl [r13] = r35,1
      br.ctop loop
  
```

Memory

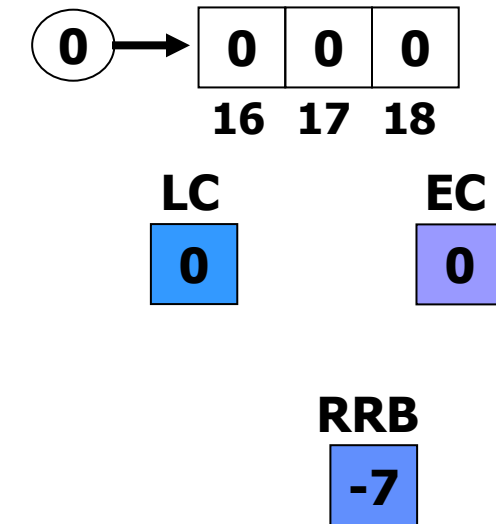


General Registers (Physical)



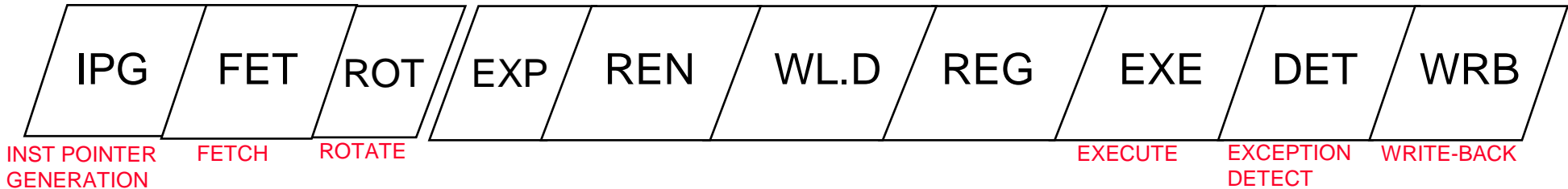
General Registers (Logical)

Predicate Registers

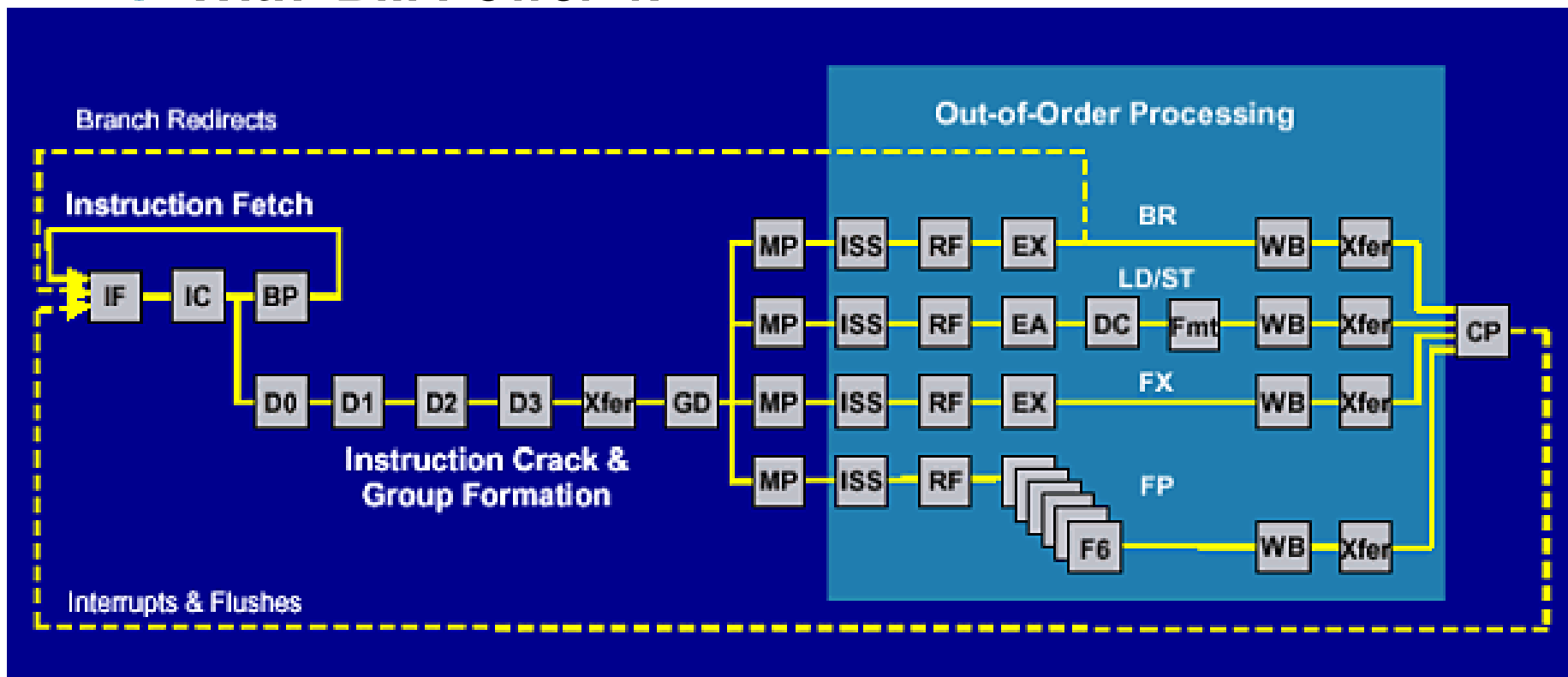


Comments on Itanium

Compare Itanium II



With IBM Power 4:



Top 20 SPEC systems

Top 20 SPECint2000

Top 20 SPECfp2000

#	MHz	Processor	int peak	int base	Full results	MHz	Processor	fp peak	fp base	Full results
1	2933	Core 2 Duo EE	3119	3108	HTML	2300	POWER5+	3642	3369	HTML
2	3000	Xeon 51xx	3102	3089	HTML	1600	DC Itanium 2	3098	3098	HTML
3	2666	Core 2 Duo	2848	2844	HTML	3000	Xeon 51xx	3056	2811	HTML
4	2660	Xeon 30xx	2835	2826	HTML	2933	Core 2 Duo EE	3050	3048	HTML
5	3000	Opteron	2119	1942	HTML	2660	Xeon 30xx	3044	2763	HTML
6	2800	Athlon 64 FX	2061	1923	HTML	1600	Itanium 2	3017	3017	HTML
7	2800	Opteron AM2	1960	1749	HTML	2667	Core 2 Duo	2850	2847	HTML
8	2300	POWER5+	1900	1820	HTML	1900	POWER5	2796	2585	HTML
9	3733	Pentium 4 E	1872	1870	HTML	3000	Opteron	2497	2260	HTML
10	3800	Pentium 4 Xeon	1856	1854	HTML	2800	Opteron AM2	2462	2230	HTML
11	2260	Pentium M	1839	1812	HTML	3733	Pentium 4 E	2283	2280	HTML
12	3600	Pentium D	1814	1810	HTML	2800	Athlon 64 FX	2261	2086	HTML
13	2167	Core Duo	1804	1796	HTML	2700	PowerPC 970MP	2259	2060	HTML
14	3600	Pentium 4	1774	1772	HTML	2160	SPARC64 V	2236	2094	HTML
15	3466	Pentium 4 EE	1772	1701	HTML	3730	Pentium 4 Xeon	2150	2063	HTML
16	2700	PowerPC 970MP	1706	1623	HTML	3600	Pentium D	2077	2073	HTML
17	2600	Athlon 64	1706	1612	HTML	3600	Pentium 4	2015	2009	HTML
18	2000	Pentium 4 Xeon LV	1668	1663	HTML	2600	Athlon 64	1829	1700	HTML
19	2160	SPARC64 V	1620	1501	HTML	1700	POWER4+	1776	1642	HTML
20	1600	Itanium 2	1590	1590	HTML	3466	Pentium 4 EE	1724	1719	HTML

With Auto-parallelisation

Top 20 SPECint2000

Top 20 SPECfp2000

#	MHz	Processor	int peak	int base	Full results	MHz	Processor	fp peak	fp base	Full results
1						2100	POWER5+	4051	3210	HTML
2						3000	Opteron	3538	2851	HTML
3						2600	Opteron AM2	3338	2711	HTML
4						1200	UltraSPARC III Cu	1344	1074	HTML

**Aces Hardware
analysis of SPEC
benchmark data
[http://www.aces
hardware.com/S
PECmine/top.jsp](http://www.aceshardware.com/SPECmine/top.jsp)
(ca.2007)**

<http://www.spec.org/cpu2006/results/cpu2006.html>

Summary#1: Hardware versus Software Speculation Mechanisms

- ✦ **To speculate extensively, must be able to disambiguate memory references**
 - ➡ Much easier in HW than in SW for code with pointers
- ✦ **HW-based speculation works better when control flow is unpredictable, and when HW-based branch prediction is superior to SW-based branch prediction done at compile time**
 - ➡ Mispredictions mean wasted speculation
- ✦ **HW-based speculation maintains precise exception model even for speculated instructions**
- ✦ **HW-based speculation does not require compensation or bookkeeping code**

Summary#2: Hardware versus Software Speculation Mechanisms cont' d

- Compiler-based approaches may benefit from the ability to see further in the code sequence, resulting in better code scheduling

- HW-based speculation with dynamic scheduling does not require different code sequences to achieve good performance for different implementations of an architecture

 - ➡ may be the most important in the long run?

- Example: ARM's "big.LITTLE" architecture

 - ➡ Multicore processor with a mixture of large out-of-order cores (A15) and small in-order cores (A7) (eg Exynos 5 Octa in Samsung Galaxy S4)


 - ➡ Compiler is configured to schedule for in-order, assuming the out-of-order processor is less sensitive to instruction scheduling


Extra slides for interest/fun

Associativity in floating point

✚ $(a+b)+c = a+(b+c)$?

✚ Example: Consider 3-digit base-10 floating-point

$1+1+1+1+1+1+1+1+\dots+1+1+1+1+1+1+1+1+1+1+1+1+1000$

 1000 ones

$1000+1+1+1+1+1+1+1+1+\dots+1+1+1+1+1+1+1+1+1+1+1+1$

 1000 ones

Consequence: many compilers use loop unrolling and reassociation to enhance parallelism in summations

And results are different!

But you can tell the compiler not to, eg:

“-fp-model precise” with Intel’s compilers

✚ (What’s the right way to sum an array? See

➡ http://en.wikipedia.org/wiki/Kahan_summation_algorithm)

- In the example processor that can only execute one instruction per cycle, unrolling is important because the loop control instructions become the critical factor.
- In machines that can issue multiple instructions per cycle, this is likely not the case - there are opportunities to issue some instructions "for free" if you can schedule them into unused issue slots.
- In that case, software pipelining should lead to better performance than unrolling, though the difference might be small with a sufficiently-high unroll factor.
- You might also consider the energy cost: unrolling means we cache and store more instructions. But software pipelining without unrolling means we execute more loop-control instructions.
- Obviously if loop unrolling were to lead to instruction-cache misses, that'd be bad.
- So actually, the optimum strategy is likely to be a hybrid.

- This is actually only the beginning. You can sometimes do better by unrolling an *outer* loop - this is called "unroll and jam", because we unroll the outer loop to produce two copies of the inner loop, then we jam them together. Consider matrix-matrix multiply (again!):

```
for(i=0; i<4; i++)
  for(j=0; j<4; j++) {
    c[i][j] = 0;
    for(k=0; k<4; k++)
      c[i][j] = a[i][k]*b[k][j]+c[i][j];
  }
```

- This has limited parallelism due to the (loop-carried dependence involved in the) summation into $C[i][j]$. After unroll-and-jam of the j -loop by 1, we have:

```
for(i=0; i<4; i++)
  for(j=0; j<4; j+=2) {
    c[i][j] = 0;
    c[i][j+1] = 0;
    for(k=0; k<4; k++) {
      c[i][j]=a[i][k]*b[k][j]+c[i][j];
      c[i][j+1]=a[i][k]*b[k][j+1]+c[i][j+1];
    }
  }
```

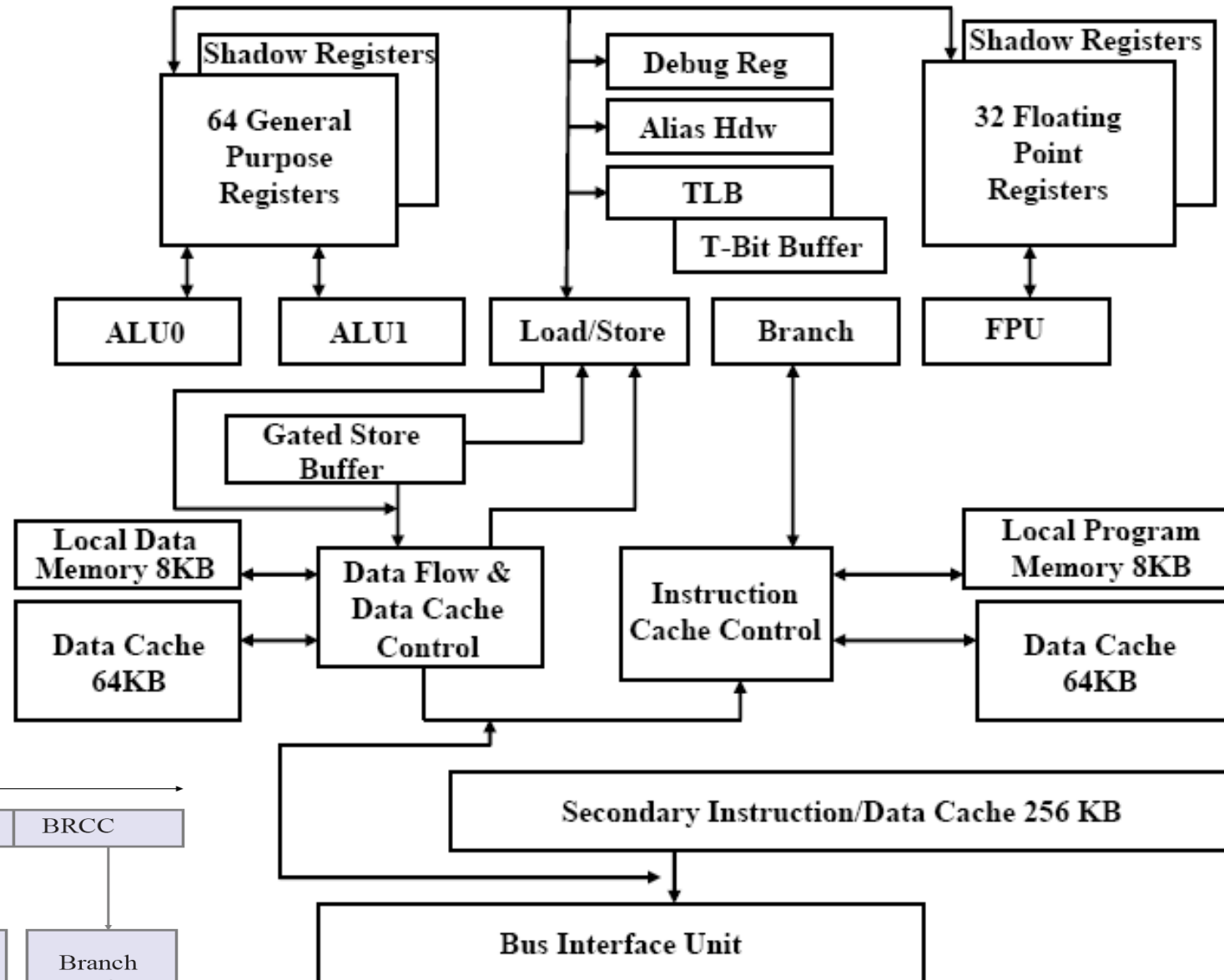
Unrolling versus software pipelining, and unroll-and-jam

- Now the inner loop has two summations to do, which are independent from one another. So it's more likely that you can fill the schedule more tightly.
- This example is taken from: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1.9319&rep=rep1&type=pdf>

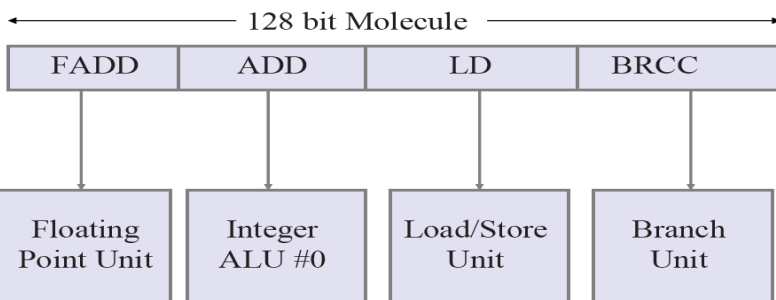
VLIW example: Transmeta Crusoe

Transmeta's Crusoe was a 5—issue VLIW processor

Instructions were dynamically translated from x86 in firmware “Code Morphing”



Note hardware support for speculation



Instruction encoding