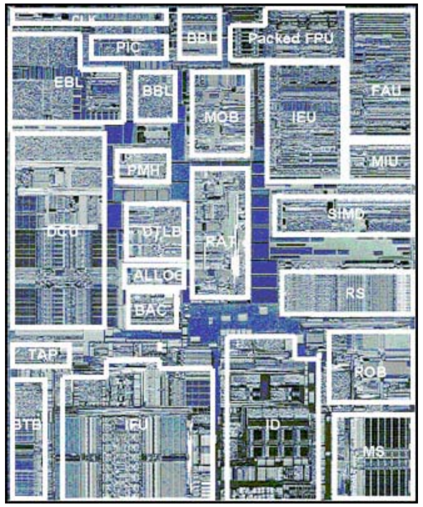
332
Advanced Computer Architecture
Chapter 5

Instruction Level Parallelism - the static scheduling approach

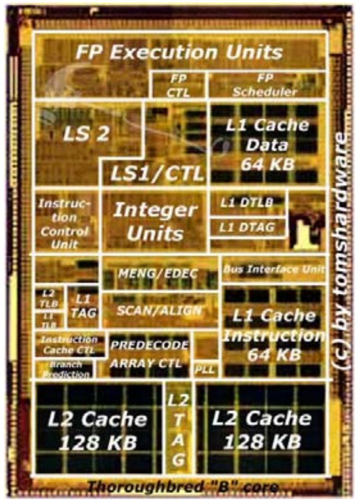
March 2006

Paul H J Kelly

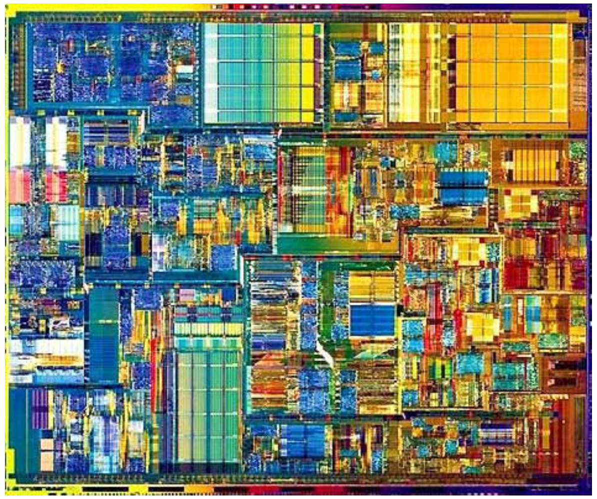
These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (3rd ed)*, and on the lecture slides of David Patterson and John Kubiatowicz's Berkeley course



- Intel Pentium III



- AMD Athlon CPU core



- Pentium 4 "Netburst"

- Intel® Pentium® 4 and Intel® Xeon™ Processor Optimization
  - http://www.intel.com/design/pentium4/manuals/24896607.pdf
- **Desktop Performance & Optimization for Intel® Pentium® 4 Processor**
  - ftp://download.intel.com/design/pentium4/papers/24943801.pdf **(much shorter!)**
- Intel® compilers
  - http://www.intel.com/software/products/compilers/
- Intel's VTune performance analysis tool
  - http://www.intel.com/software/products/vtune/
- AMD Athlon x86 Code Optimization Guide
  - http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf
  - (see page 67 for the nine-step development of a routine to copy an array – 570MB/s to 1630MB/s)

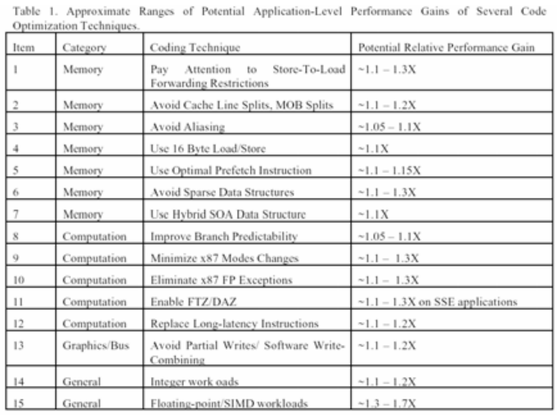


Table 1. Approximate Ranges of Potential Application-Level Performance Gains of Several Code Optimization Techniques.

Item	Category	Coding Technique	Potential Relative Performance Gain
1	Memory	Pay Attention to Store-To-Load Forwarding Restrictions	~1.1 – 1.3X
2	Memory	Avoid Cache Line Splits, MOB Splits	~1.1 – 1.2X
3	Memory	Avoid Aliasing	~1.05 – 1.1X
4	Memory	Use 16 Byte Load/Store	~1.1X
5	Memory	Use Optimal Prefetch Instruction	~1.1 – 1.15X
6	Memory	Avoid Sparse Data Structures	~1.1 – 1.3X
7	Memory	Use Hybrid SOA Data Structure	~1.1X
8	Computation	Improve Branch Predictability	~1.05 – 1.1X
9	Computation	Minimize x87 Modes Changes	~1.1 – 1.3X
10	Computation	Eliminate x87 FP Exceptions	~1.1 – 1.3X
11	Computation	Enable FTZ/DAZ	~1.1 – 1.3X on SSE applications
12	Computation	Replace Long-latency Instructions	~1.1 – 1.2X
13	Graphics/Bus	Avoid Partial Writes/ Software Write-Combining	~1.1 – 1.2X
14	General	Integer work oads	~1.1 – 1.2X
15	General	Floating-point/SIMD workloads	~1.3 – 1.7X

Example: Pentium 4 memory aliasing

- "There are several cases where addresses with a given stride will compete for some resource in the memory hierarchy. Note that first-level cache lines are 64 bytes and second-level cache lines are 128 bytes. Thus the least significant 6 or 7 bits are not considered in alias comparisons. The aliasing cases are listed below.
  - 2K for data – map to the same first-level cache set (32 sets, 64-byte lines). There are 4 ways in the first-level cache, so if there are more that 4 lines that alias to the same 2K modulus in the working set, there will be an excess of first-level cache misses.
  - 16K for data – will look same to the store-forwarding logic. If there has been a store to an address which aliases with the load, the load will stall until the store data is available.
  - 16K for code – can only be one of these in the trace cache at a time. If two traces whose starting addresses are 16K apart are in the same working set, the symptom will be a high trace cache miss rate. Solve this by offsetting one of the addresses by 1 or more bytes.
  - 32K for code or data – map to the same second-level cache set (256 sets, 128-byte lines). There are 8 ways in the second-level cache, so if there are more than 8 lines that alias to the same 32K modulus in the working set, there will be an excess of second-level cache misses.
  - 64K for data – can only be one of these in the first-level cache at a time. If a reference (load or store) occurs that has bits 0-15 of the linear address, which are identical to a reference (load or store) which is underway, then second reference cannot begin until first one is kicked out of cache. Avoiding this kind of aliasing can lead to a factor of three speedup." (http://www.intel.com/design/pentium4/manuals/24896607.pdf page 2-38)

Review: extreme dynamic ILP

- P6 (Pentium Pro, II, III, AMD Athlon)
  - Translate most 80x86 instructions to micro-operations
    - Longer pipeline than RISC instructions
  - Dynamically execute micro-operations
- "Netburst" (Pentium 4, …)
  - Much longer pipeline, higher clock rate in same technology as P6
  - Trace Cache to capture micro-operations, avoid hardware translation
- How can we take these ideas further?
  - Complexity of issuing multiple instructions per cycle
  - And of committing them
    - n-way multi-issue processor with an m-instruction dynamic scheduling window
    - m must increase if n is increased
    - Need n register ports
    - Need to compare each of the n instruction's src and dst regs to determine dependence
  - Predicting and speculating across multiple branches
  - With many functional units and registers, wires will be long - need pipeline stage just to move the data across the chip

Overview

- The previous Chapter: Dynamic scheduling, out-of-order (o-o-o): binary compatible, exploiting ILP in hardware: BTB, ROB, Reservation Stations, ...
- 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
  - 7 years in the making?

Static Branch Prediction

- Simplest: Predict taken
  - average misprediction rate = untaken branch frequency, which for the SPEC programs is 34%.
  - Unfortunately, the misprediction rate ranges from not very accurate (59%) to highly accurate (9%)
- Predict on the basis of branch direction?
  - choosing backward-going branches to be taken (loop)
  - forward-going branches to be not taken (if)
  - SPEC programs, however, most forward-going branches are taken => predict taken is better
- Predict branches on the basis of profile information collected from earlier runs
  - Misprediction varies from 5% to 22%

Running Example

- This code adds a scalar to a vector:

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

- Assume following latency all examples

Instruction producing result	*Instruction using result*	*Execution in cycles*	*Latency in cycles*
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	1	1
Load double	Store double	1	0
Integer op	Integer op	1	0

FP Loop: Where are the Hazards?

- First translate into MIPS code:
  - -To simplify, assume 8 is lowest address

```
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?

FP Loop 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
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
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 FP Loop Minimizing 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 move past DSUBUI
```

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

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
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 Loop Four Times (straightforward way)

```
1 Loop:L.D    F0,0(R1)        1 cycle stall
2      ADD.D  F4,F0,F2        2 cycles stall
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
```

Rewrite loop to minimize stalls?

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration

Assumes R1 is multiple of 4

Unrolled Loop Detail

- Do not usually know upper bound of loop
- Suppose it is n, and we would like to unroll the loop to make k copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
  - 1st executes (n mod k) times and has a body that is the original loop
  - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
  - For large values of n, most of the execution time will be spent in the unrolled loop

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
```

14 clock cycles, or 3.5 per iteration

- 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 do such changes?

Compiler Perspectives on Code Movement

- Compiler concerned about dependencies in program
- Whether or not a HW hazard depends on pipeline
- Try to schedule to avoid hazards that cause performance losses
- (True) Data dependencies (RAW if a hazard for HW)
  - Instruction i produces a result used by instruction j, or
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
- If dependent, can't execute in parallel
- Easy to determine for registers (fixed names)
- Hard for memory ("memory disambiguation" problem):
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?

Where are the name dependencies?

```
Loop:L.D    F0,0(R1)
     ADD.D  F4,F0,F2
     S.D    0(R1),F4     ;drop DSUBUI & BNEZ
     L.D    F0,-8(R1)
     ADD.D  F4,F0,F2
     S.D    -8(R1),F4    ;drop DSUBUI & BNEZ
     L.D    F0,-16(R1)
     ADD.D  F4,F0,F2
     S.D    -16(R1),F4   ;drop DSUBUI & BNEZ
     L.D    F0,-24(R1)
     ADD.D  F4,F0,F2
     S.D    -24(R1),F4
     DSUBUI R1,R1,#32    ;alter to 4*8
     BNEZ   R1,LOOP
     NOP
```

How can remove them?

Where are the name dependencies?

```
Loop:L.D    F0,0(R1)
     ADD.D  F4,F0,F2
     S.D    0(R1),F4     ;drop DSUBUI & BNEZ
     L.D    F6,-8(R1)
     ADD.D  F8,F6,F2
     S.D    -8(R1),F8    ;drop DSUBUI & BNEZ
     L.D    F10,-16(R1)
     ADD.D  F12,F10,F2
     S.D    -16(R1),F12  ;drop DSUBUI & BNEZ
     L.D    F14,-24(R1)
     ADD.D  F16,F14,F2
     S.D    -24(R1),F16
     DSUBUI R1,R1,#32    ;alter to 4*8
     BNEZ   R1,LOOP
     NOP
```

The original "register renaming"

Compiler Perspectives on Code Movement

- Name Dependencies are Hard to discover for Memory Accesses
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?
- Our example required compiler to know that if R1 doesn't change then:

  0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)

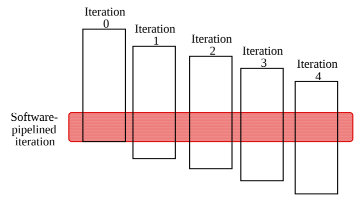
  There were no dependencies between some loads and stores so they could be moved by each other

Steps Compiler Performed to Unroll

- Check OK to move the S.D after DSUBUI and BNEZ, and find amount to adjust S.D offset
- Determine unrolling the loop would be useful by finding that the loop iterations were independent
- Rename registers to avoid name dependencies
- Eliminate extra test and branch instructions and adjust the loop termination and iteration code
- Determine loads and stores in unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent
  - requires analyzing memory addresses and finding that they do not refer to the same address.
- Schedule the code, preserving any dependences needed to yield same result as the original code

Another possibility: 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 SW)



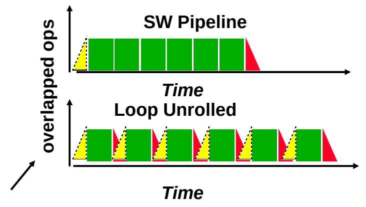
Software Pipelining Example

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
```

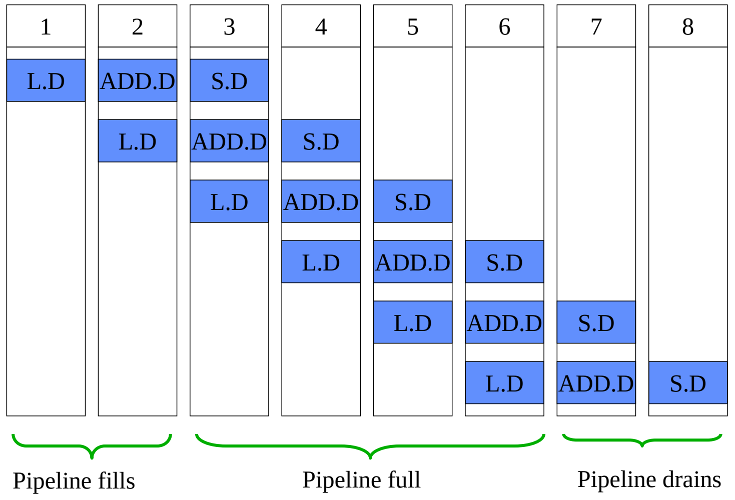
After: Software Pipelined

```
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



1	2	3	4	5	6	7	8
L.D	ADD.D	S.D					
	L.D	ADD.D	S.D				
		L.D	ADD.D	S.D			
			L.D	ADD.D	S.D		
				L.D	ADD.D	S.D	
					L.D	ADD.D	S.D

When Safe to Unroll Loop?

- Example: Where are data dependencies? (A,B,C distinct & non-overlapping)

```
   for (i=0; i<100; i=i+1) {
          A[i+1] = A[i] + C[i];    /* S1 */
          B[i+1] = B[i] + A[i+1];  /* S2 */
   }
```

1. S2 uses the value, A[i+1], computed by S1 in the same iteration.

2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].
This is a "loop-carried dependence": between iterations

- For our prior example, each iteration was distinct
- Implies that iterations can't be executed in parallel, Right????

Does a loop-carried dependence mean there is no parallelism???

- Consider:

```
    for (i=0; i< 8; i=i+1) {
          A = A + C[i];     /* S1 */
    }
```

Could compute:

```
"Cycle 1":  temp0 = C[0] + C[1];
            temp1 = C[2] + C[3];
            temp2 = C[4] + C[5];
            temp3 = C[6] + C[7];
"Cycle 2":  temp4 = temp0 + temp1;
            temp5 = temp2 + temp3;
"Cycle 3":  A = temp4 + temp5;
```

- Relies on associative nature of "+".
- See "Parallelizing Complex Scans and Reductions" by Allan Fisher and Anwar Ghuloum (http://doi.acm.org/10.1145/178243.178255)

Associativity in floating point

- (a+b)+c = a+(b+c) ?
- Example: Consider 3-digit base-10 floating-point

$$\underbrace{1+1+1+1+1+1+1+1+\ldots\ldots+1+1+1+1+1+1+1+1+1+1+1}_{\text{1000 ones}}+1000$$

$$1000+\underbrace{1+1+1+1+1+1+1+1+1+\ldots\ldots+1+1+1+1+1+1+1+1+1+1+1+1}_{\text{1000 ones}}$$

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

And results are different!

Hardware Support for Exposing More Parallelism at Compile-Time

- Conditional or Predicated Instructions
  - Discussed before in context of branch prediction
  - Conditional instruction execution
- First instruction slot   Second instruction slot

```
LW    R1,40(R2)        ADD R3,R4,R5
                       ADD R6,R3,R7
BEQZ  R10,L
LW    R8,0(R10)
LW    R9,0(R8)
```

- Waste slot since 3rd LW dependent on result of 2nd LW

Hardware Support for Exposing More Parallelism at Compile-Time

- Use predicated version load word (LWC)?
  - load occurs unless the third operand is 0
- First instruction slot   Second instruction slot

```
LW    R1,40(R2)         ADD R3,R4,R5
LWC     R8,20(R10),R10  ADD R6,R3,R7
BEQZ  R10,L
LW    R9,0(R8)
```

- If the sequence following the branch were short, the entire block of code might be converted to predicated execution, and the branch eliminated

Exception Behavior Support

- Several mechanisms to ensure that speculation by compiler does not violate exception behavior
  - For example, cannot raise exceptions in predicated code if annulled
  - Prefetch does not cause exceptions

Hardware Support for Memory Reference Speculation

- To help compiler to move loads across stores, when it cannot be absolutely certain that such a movement is correct, a special instruction to check for address conflicts can be included in the architecture
  - The special instruction is left at the original location of the load and the load is moved up across stores
  - When a speculated load is executed, the hardware saves the address of the accessed memory location
  - If a subsequent store changes the location before the check instruction, then the speculation has failed
  - If only load instruction was speculated, then it suffices to redo the load at the point of the check instruction

What if We Can Change Instruction Set?

- Superscalar processors decide on the fly how many instructions to issue
  - HW complexity of Number of instructions to issue $O(n^2)$
- Why not allow compiler to schedule instruction level parallelism explicitly?
- Format the instructions in a potential issue packet so that HW need not check explicitly for dependences

VLIW: Very Large Instruction Word

- 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)
- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => 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
  - 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

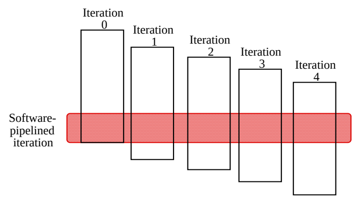
Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
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 in SS)

Recall: Software Pipelining

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



Recall: Software Pipelining Example

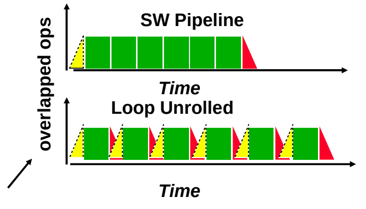
Before: Unrolled 3 times

```
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



Software Pipelining with Loop Unrolling in VLIW

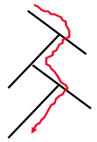
Memory reference 1	Memory reference 2	FP operation 1	FP op. 2	Int. op/ branch	Clock
L.D F0,-48(R1)	ST 0(R1),F4	ADD.D F4,F0,F2			1
L.D F6,-56(R1)	ST -8(R1),F8	ADD.D F8,F6,F2		DSUBUI R1,R1,#24	2
L.D F10,-40(R1)	ST 8(R1),F12	ADD.D F12,F10,F2		BNEZ R1,LOOP	3

- Software pipelined across 9 iterations of original loop
  - In each iteration of above loop, we:
    - Store to m,m-8,m-16 (iterations I-3,I-2,I-1)
    - Compute for m-24,m-32,m-40 (iterations I,I+1,I+2)
    - Load from m-48,m-56,m-64 (iterations I+3,I+4,I+5)
- 9 results in 9 cycles, or 1 clock per iteration
- Average: 3.3 ops per clock, 66% efficiency

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

Trace Scheduling

- Parallelism across IF branches vs. LOOP branches?
- Two steps:
  - *Trace Selection*
    - Find likely sequence of basic blocks (*trace*) of (statically predicted or profile predicted) long sequence of straight-line code
  - *Trace Compaction*
    - Squeeze trace into few VLIW instructions
    - Need bookkeeping code in case prediction is wrong
- This is a form of compiler-generated speculation
  - Compiler must generate "fixup" code to handle cases in which trace is not the taken branch
  - Needs extra registers: undoes bad guess by discarding
- Subtle compiler bugs mean wrong answer vs. poorer performance; no hardware interlocks



Advantages of HW (Tomasulo) vs. SW (VLIW) Speculation

- HW advantages:
  - HW better at memory disambiguation since knows actual addresses
  - HW better at branch prediction since lower overhead
  - HW maintains precise exception model
  - HW does not execute bookkeeping instructions
  - Same software works across multiple implementations
  - Smaller code size (not as many nops filling blank instructions)
- SW advantages:
  - Window of instructions that is examined for parallelism much higher
  - Much less hardware involved in VLIW (unless you are Intel…!)
  - More involved types of speculation can be done more easily
  - Speculation can be based on large-scale program behavior, not just local information

Superscalar v. VLIW

- Smaller code size
- Binary compatibility across generations of hardware

- Simplified Hardware for decoding, issuing instructions
- No Interlock Hardware (compiler checks?)
- More registers, but simplified Hardware for Register Ports (multiple independent register files?)

Problems with First Generation VLIW

- Increase in code size
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding
- Operated in lock-step; no hazard detection HW
  - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might know functional unit latencies, but caches harder to predict
- Binary code compatibility
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code

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

- IA-64: instruction set architecture; EPIC is type
  - EPIC = 2nd generation VLIW?
- Itanium™ first implementation
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline
  - Not competitive
- Itanium 2
  - 6-wide, 8-stage pipeline
  - 16KB L1I, 16KB L1D (one cycle), 256KB L2 (5 cycle), 3MB L3 (12 cycle), all on-die
  - http://www.intel.com/products/server/processors/server/itanium2/
  - Competitive for some applications (eg SPEC FP)
- 128 64-bit integer registers + 128 82-bit floating point registers
  - Not separate register files per functional unit as in old VLIW
- Hardware checks dependencies (interlocks => binary compatibility over time)
- Predicated execution (select 1 out of 64 1-bit flags) => 40% fewer mispredictions?

IA-64 Registers

- The integer registers are configured to help accelerate procedure calls using a register stack
  - mechanism similar to that developed in the Berkeley RISC-I processor and used in the SPARC architecture.
  - Registers 0-31 are always accessible and addressed as 0-31
  - 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
- 8 64-bit Branch registers used to hold branch destination addresses for indirect branches
- 64 1-bit predicate registers

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

- Instruction group: a sequence of consecutive instructions with no register data dependences
  - All the instructions in a group could be executed in parallel, if sufficient hardware resources existed and if any dependences through memory were preserved
  - An instruction group can be arbitrarily long, but the compiler must explicitly indicate the boundary between one instruction group and another by placing a stop between 2 instructions that belong to different groups
- 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
- 3 Instructions in 128 bit "groups"; field determines if instructions dependent or independent
  - Smaller code size than old VLIW, larger than x86/RISC
  - Groups can be linked to show independence > 3 instr

5 Types of Execution in Bundle

Execution Unit Slot	*Instruction type*	*Instruction Description*	*Example Instructions*
I-unit	A	Integer ALU	add, subtract, and, or, cmp
	I	Non-ALU Int	shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, cmp
	M	Memory access	Loads, stores for int/FP regs
F-unit	F	Floating point	Floating point instructions
B-unit	B	Branches	Conditional branches, calls
L+X	L+X	Extended	Extended immediates, stops

• 5-bit template field within each bundle describes both the presence of any stops associated with the bundle *and* the execution unit type required by each instruction within the bundle (see Fig 4.12 page 271)

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 the SW-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

The concept of a software pipelining branch:

```
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.

- Register rotation is useful for procedure calls
- It's also useful for software-pipelined loops
- 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

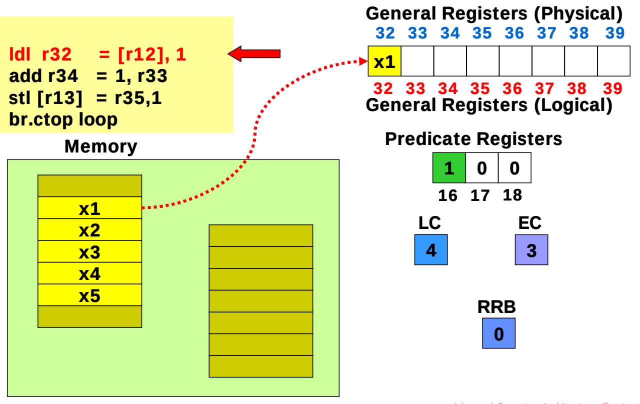
```
        mov pr.rot       = 0       // Clear all rotating predicate registers
        cmp.eq p16,p0 = r0,r0   // Set p16=1
        mov ar.lc        = 4       // Set loop counter to n-1
        mov ar.ec        = 3       // Set epilog counter to 3
              ...
loop:
(p16)   ldl  r32    = [r12], 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
```

- "Stage" predicate mechanism allows successive stages of the software pipeline to be filled on start-up and drained 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

http://www.cs.ualberta.ca/~amaral/courses/680/webslides/TF-HWSupSoftPipeline/

Software Pipelining Example in the IA-64

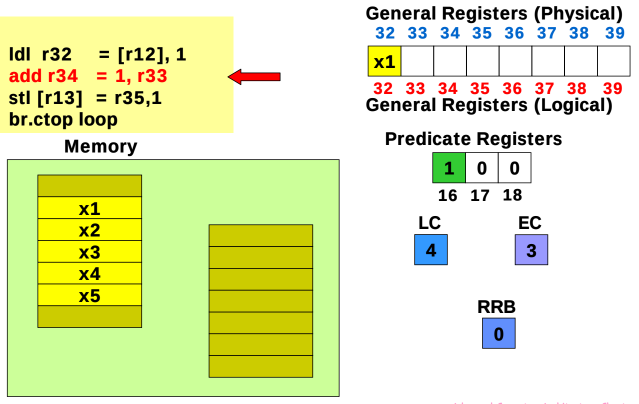
```
loop:
(p16)   ldl  r32   = [r12], 1
(p17)   add r34  = 1, r33
(p18)   stl [r13]  = r35,1
        br.ctop loop
```



http://www.cs.ualberta.ca/~amaral/courses/680/webslides/TF-HWSupSoftPipeline/

Software Pipelining Example in the IA-64

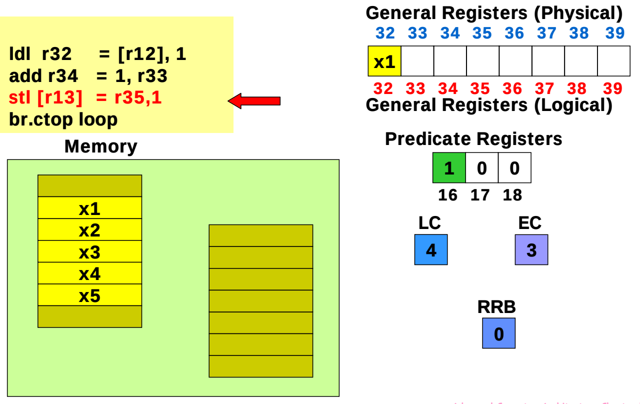
```
loop:
(p16)   ldl  r32   = [r12], 1
(p17)   add r34  = 1, r33
(p18)   stl [r13]  = r35,1
        br.ctop loop
```



http://www.cs.ualberta.ca/~amaral/courses/680/webslides/TF-HWSupSoftPipeline/

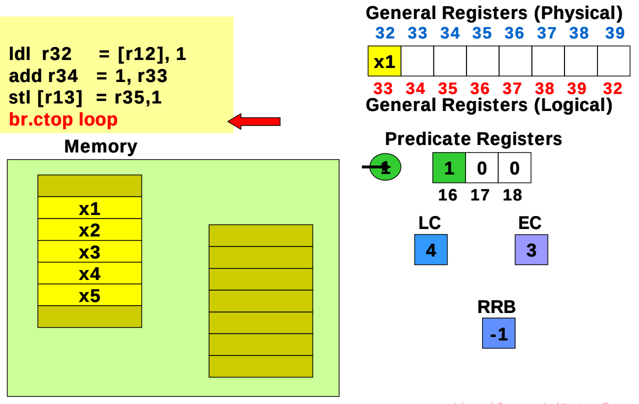
Software Pipelining Example in the IA-64

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

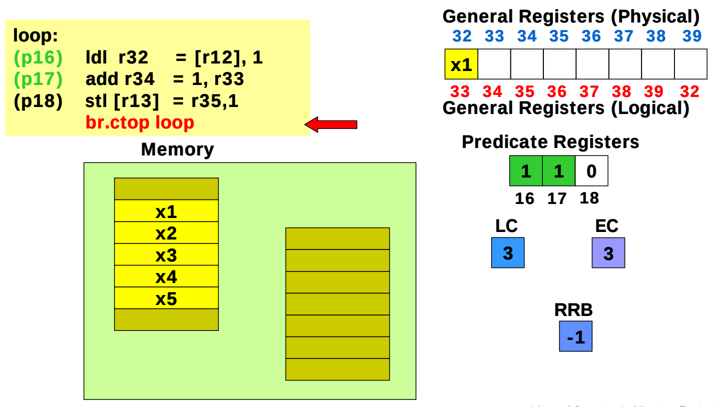


Software Pipelining Example in the IA-64

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

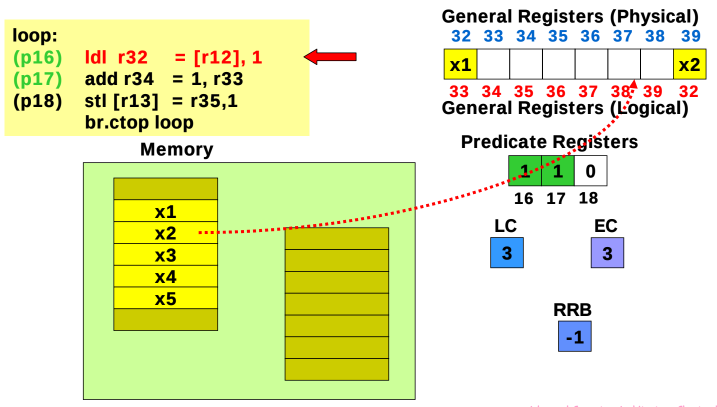


Software Pipelining Example in the IA-64



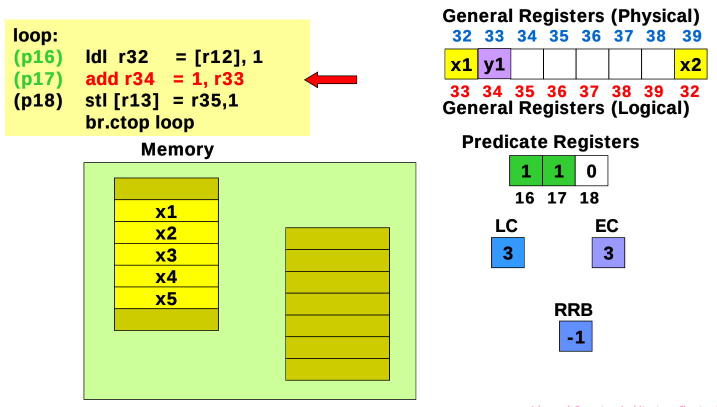
```
loop:
(p16)   ldl  r32    = [r12], 1
(p17)   add r34   = 1, r33
(p18)   stl [r13]  = r35,1
        br.ctop loop
```

Software Pipelining Example in the IA-64



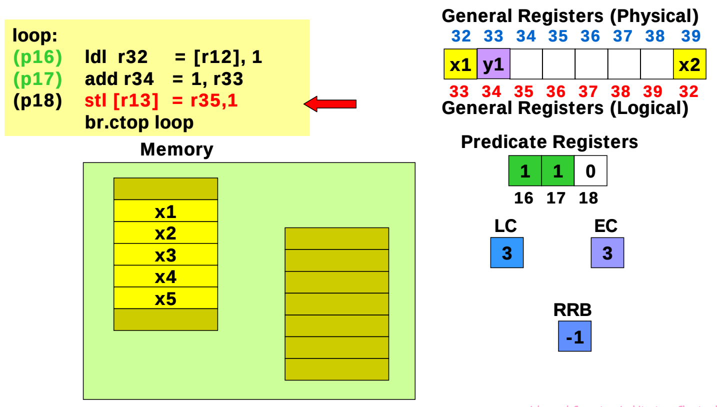
```
loop:
(p16)   ldl  r32    = [r12], 1
(p17)   add r34   = 1, r33
(p18)   stl [r13]  = r35,1
        br.ctop loop
```

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
```

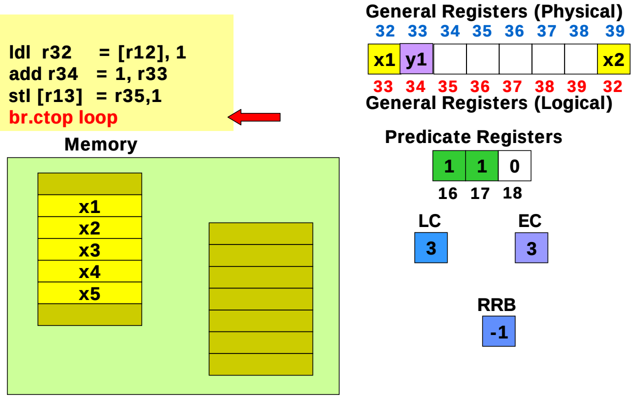
Software Pipelining Example in the IA-64



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

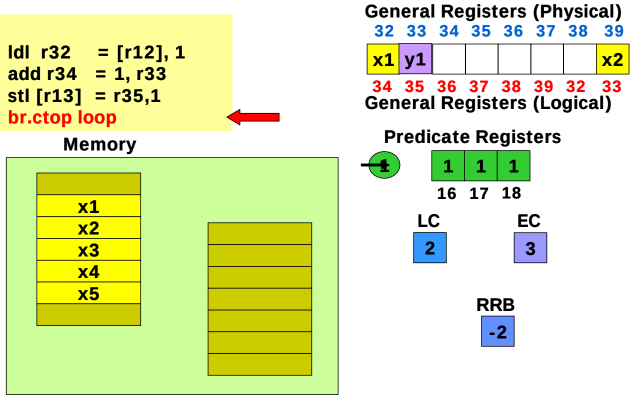
Software Pipelining Example in the IA-64

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



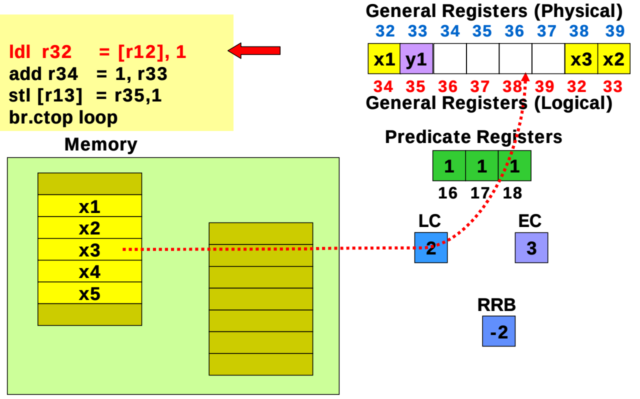
Software Pipelining Example in the IA-64

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



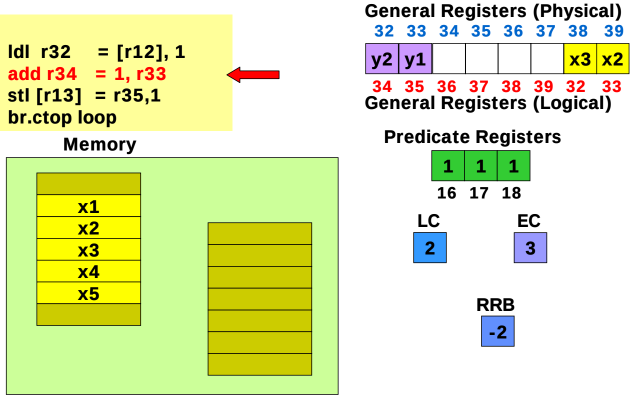
Software Pipelining Example in the IA-64

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

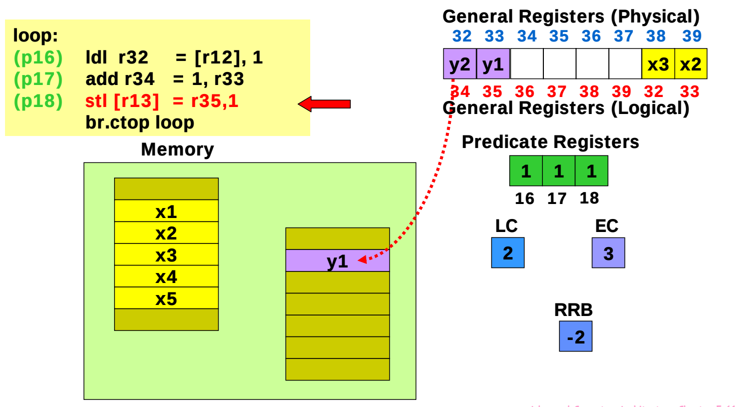


Software Pipelining Example in the IA-64

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

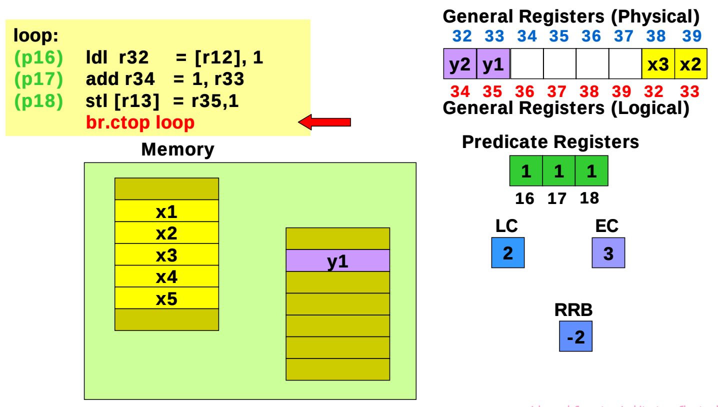


Software Pipelining Example in the IA-64



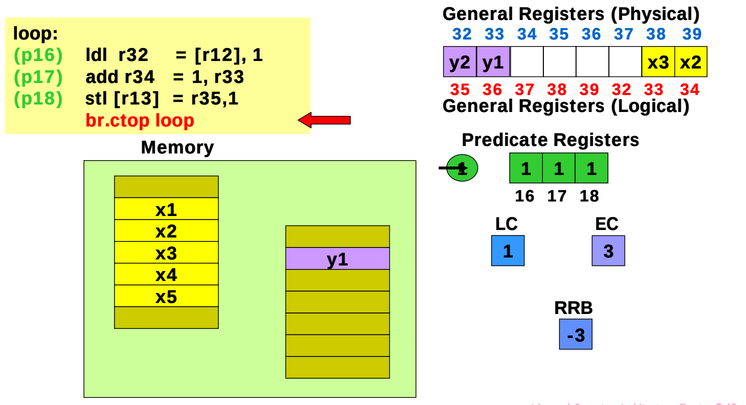
```
loop:
(p16)  ldl  r32    = [r12], 1
(p17)  add r34   = 1, r33
(p18)  stl [r13]  = r35,1
       br.ctop loop
```

Software Pipelining Example in the IA-64



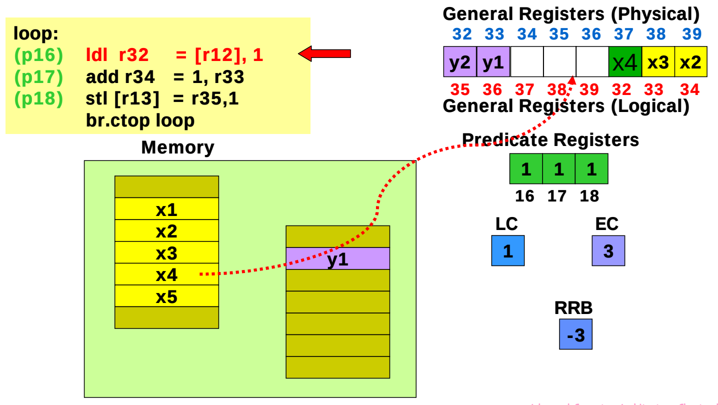
```
loop:
(p16)  ldl  r32    = [r12], 1
(p17)  add r34   = 1, r33
(p18)  stl [r13]  = r35,1
       br.ctop loop
```

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
```

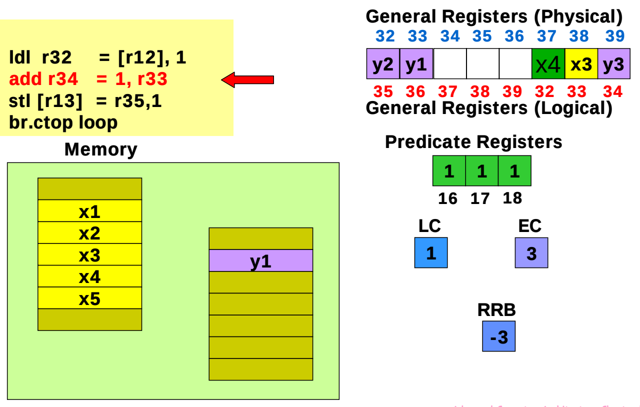
Software Pipelining Example in the IA-64



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

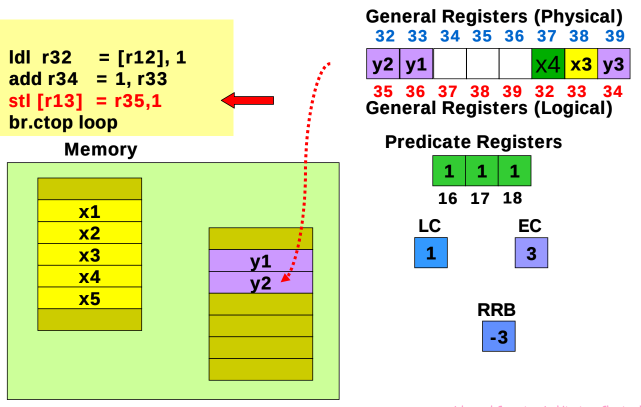
Software Pipelining Example in the IA-64

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



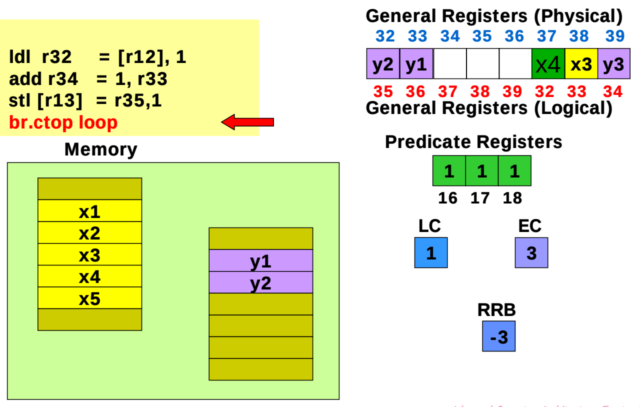
Software Pipelining Example in the IA-64

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



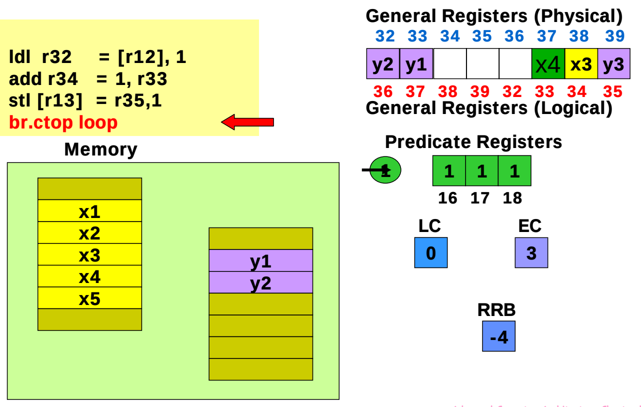
Software Pipelining Example in the IA-64

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



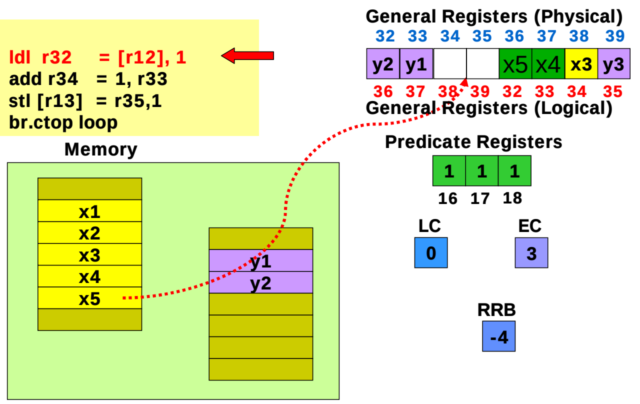
Software Pipelining Example in the IA-64

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



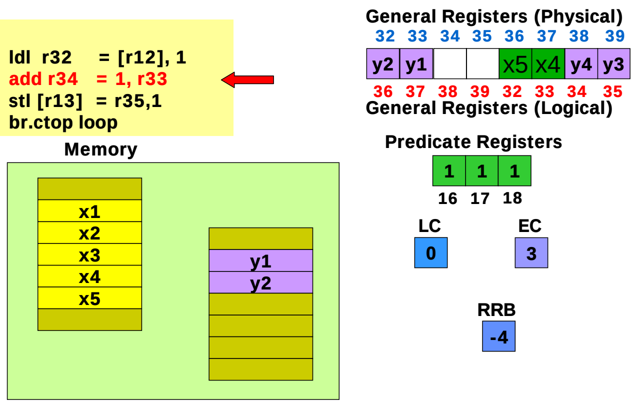
Software Pipelining Example in the IA-64

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



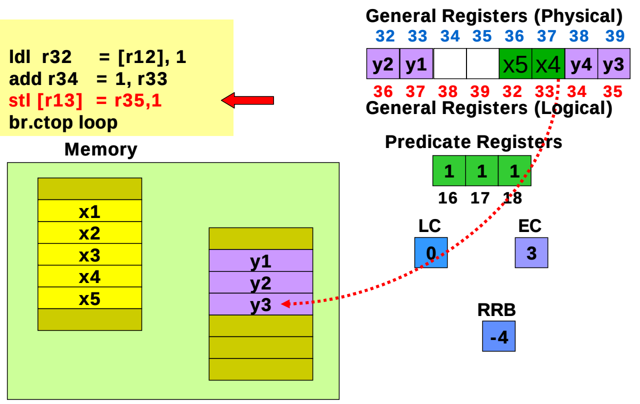
Software Pipelining Example in the IA-64

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



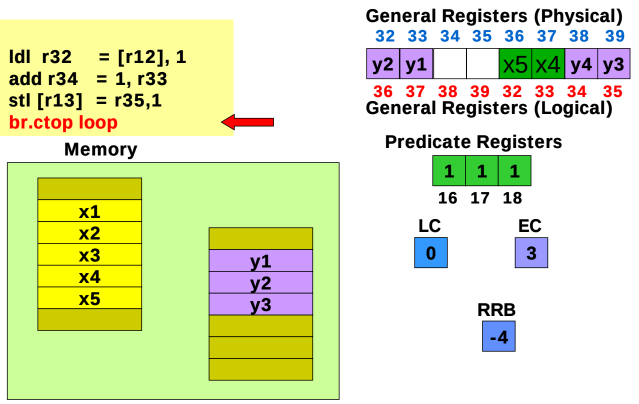
Software Pipelining Example in the IA-64

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



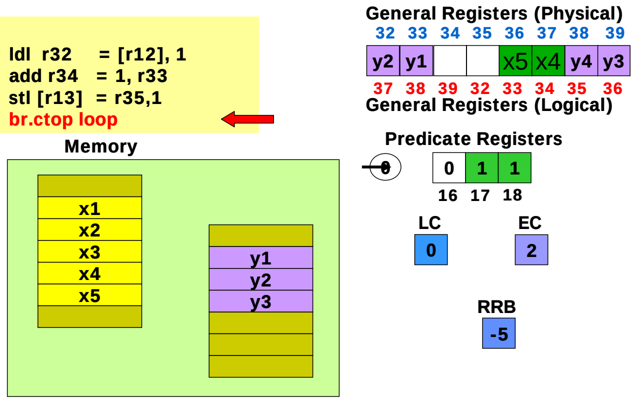
Software Pipelining Example in the IA-64

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



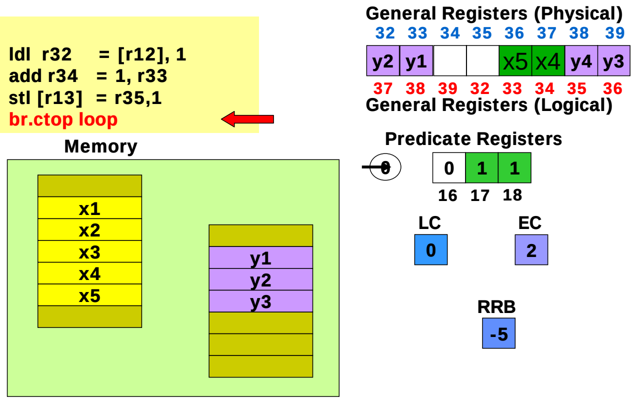
Software Pipelining Example in the IA-64

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



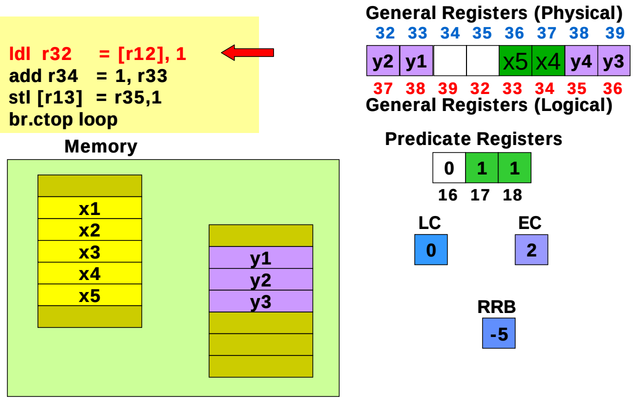
Software Pipelining Example in the IA-64

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



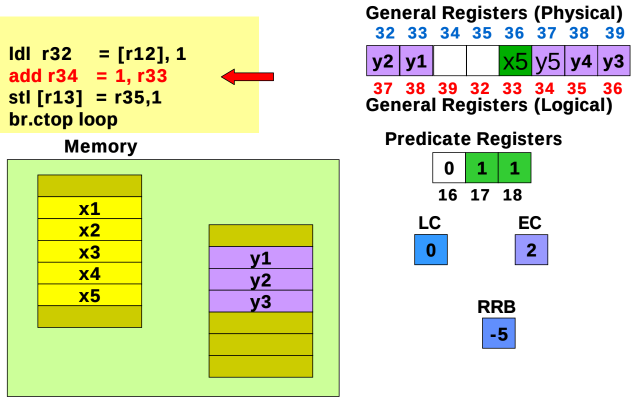
Software Pipelining Example in the IA-64

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



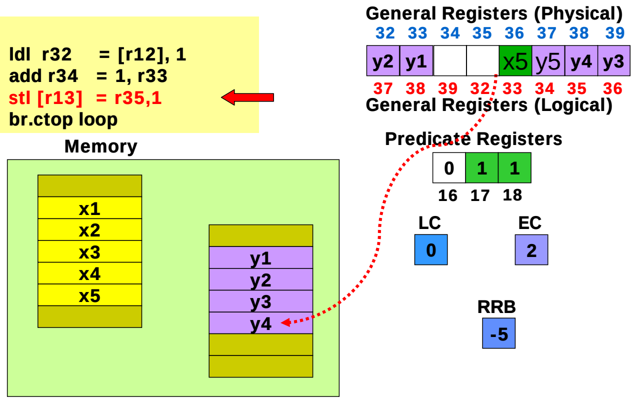
Software Pipelining Example in the IA-64

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



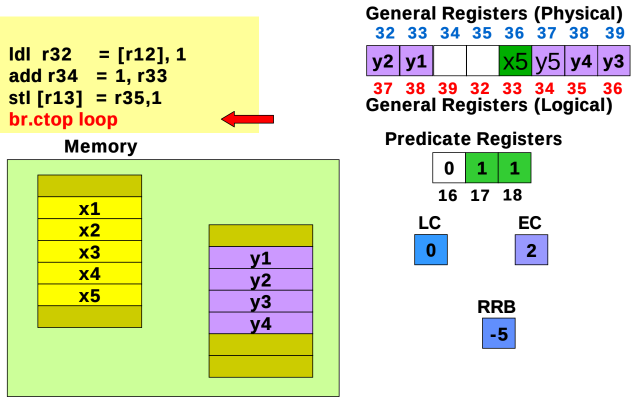
Software Pipelining Example in the IA-64

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



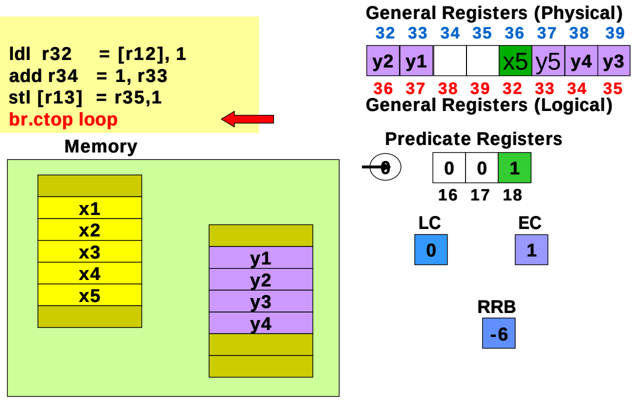
Software Pipelining Example in the IA-64

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



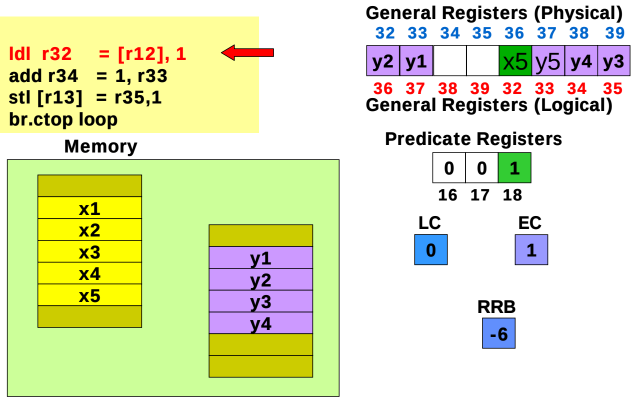
Software Pipelining Example in the IA-64

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

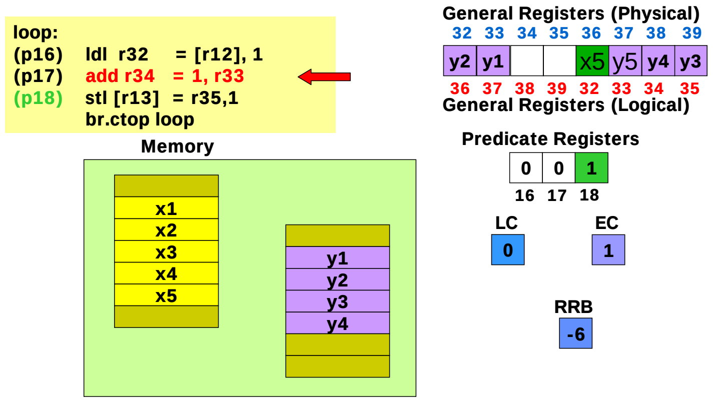


Software Pipelining Example in the IA-64

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

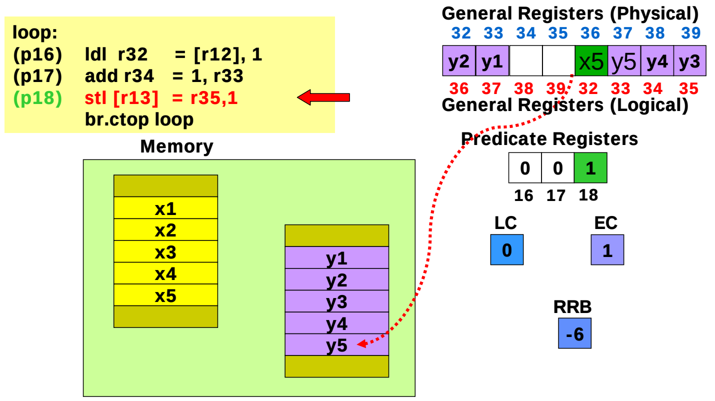


Software Pipelining Example in the IA-64



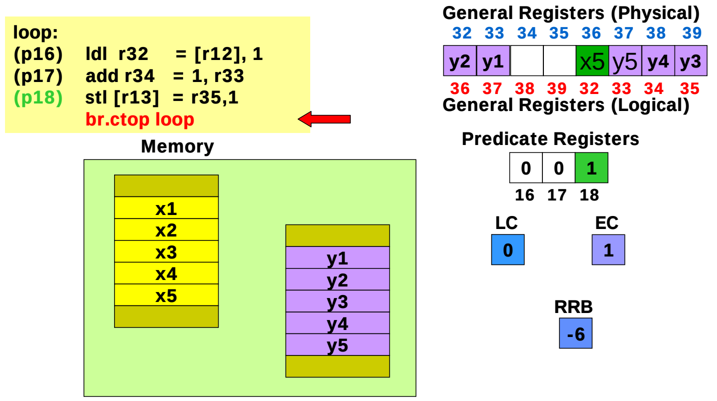
```
loop:
(p16)   ldl  r32    = [r12], 1
(p17)   add r34   = 1, r33
(p18)   stl [r13]   = r35,1
        br.ctop loop
```

Software Pipelining Example in the IA-64



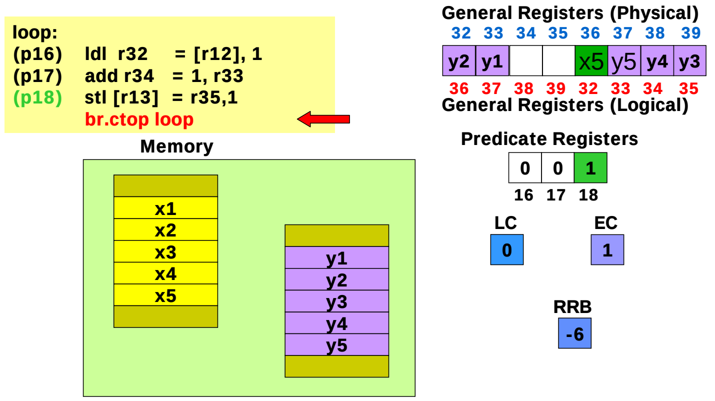
```
loop:
(p16)   ldl  r32    = [r12], 1
(p17)   add r34   = 1, r33
(p18)   stl [r13]   = r35,1
        br.ctop loop
```

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
```

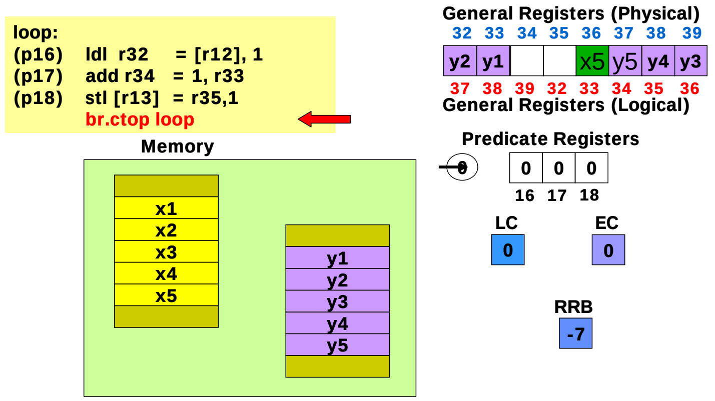
Software Pipelining Example in the IA-64



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

Software Pipelining Example in the IA-64

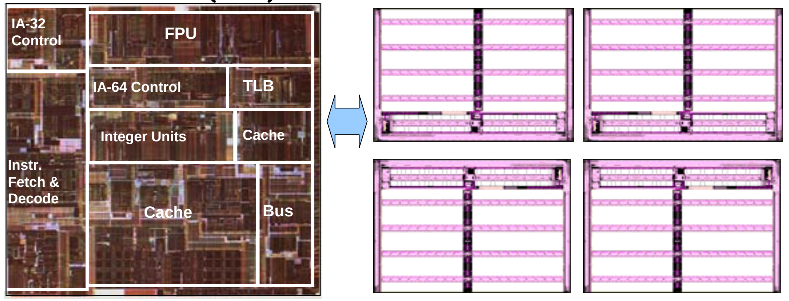
```
loop:
(p16)   ldl  r32    = [r12], 1
(p17)   add r34   = 1, r33
(p18)   stl [r13]  = r35,1
        br.ctop loop
```



Itanium™ Processor Silicon

(Copyright: Intel at Hotchips '00)

- Caches
  - 32KB L1 (2 cycle)
  - 96KB L2 (7 cycle)
  - 2 or 4 MB L3 (off chip)
- 133 MHz 64-bit bus
- SpecFP: 711
- SpecInt: 404
- 36-bit addresses (64GB)

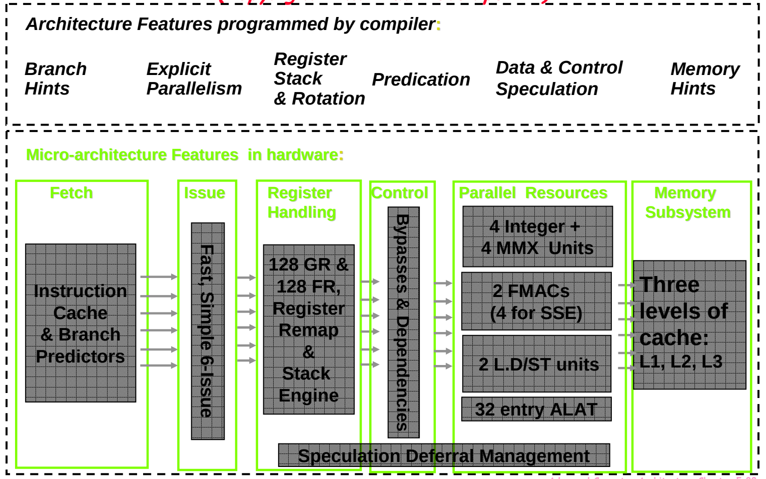


Core Processor Die

4 x 1MB L3 cache

Itanium™ EPIC Design Maximizes SW-HW Synergy

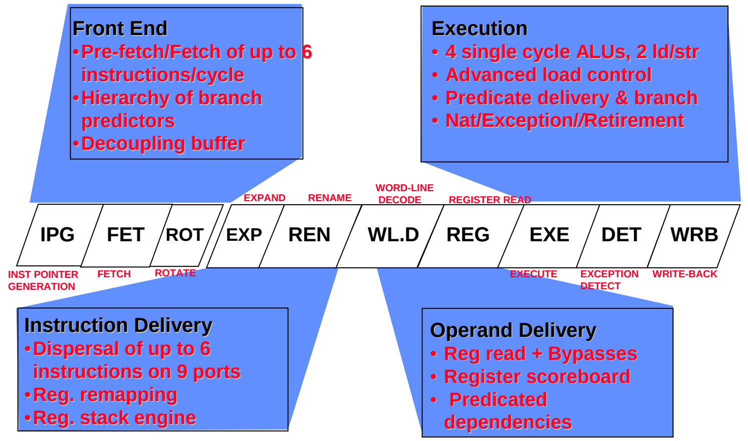
(Copyright: Intel at Hotchips '00)



Architecture Features programmed by compiler:
Branch Hints
Explicit Parallelism
Register Stack & Rotation
Predication
Data & Control Speculation
Memory Hints
Micro-architecture Features in hardware:
Fetch
Instruction Cache & Branch Predictors
Issue
Fast, Simple 6-Issue
Register Handling
128 GR & 128 FR, Register Remap & Stack Engine
Control
Bypasses & Dependencies
Parallel Resources
4 Integer + 4 MMX Units
2 FMACs (4 for SSE)
2 L.D/ST units
32 entry ALAT
Memory Subsystem
Three levels of cache: L1, L2, L3
Speculation Deferral Management

10 Stage In-Order Core Pipeline

(Copyright: Intel at Hotchips '00)

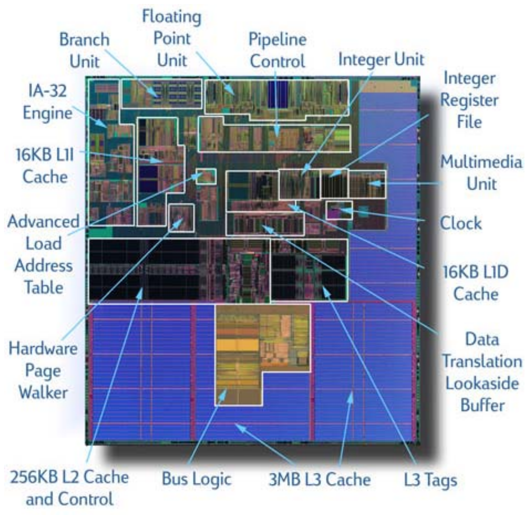


Itanium processor 10-stage pipeline

- Front-end (stages IPG, Fetch, and Rotate): prefetches up to 32 bytes per clock (2 bundles) into a prefetch buffer, which can hold up to 8 bundles (24 instructions)
  - Branch prediction is done using a multilevel adaptive predictor like P6 microarchitecture
- Instruction delivery (stages EXP and REN): distributes up to 6 instructions to the 9 functional units
  - Implements registers renaming for both rotation and register stacking.

Itanium processor 10-stage pipeline

- Operand delivery (WLD and REG): accesses register file, performs register bypassing, accesses and updates a register scoreboard, and checks predicate dependences.
  - Scoreboard used to detect when individual instructions can proceed, so that a stall of 1 instruction in a bundle need not cause the entire bundle to stall
- Execution (EXE, DET, and WRB): executes instructions through ALUs and load/store units, detects exceptions and posts NaTs, retires instructions and performs write-back
  - Deferred exception handling for speculative instructions is supported by providing the equivalent of poison bits, called NaTs for Not a Thing, for the GPRs (which makes the GPRs effectively 65 bits wide), and NaT Val (Not a Thing Value) for FPRs (already 82 bits wides)



Itanium 2

- Caches
  - 32KB L1 (1 cycle)
  - 256KB L2 (5 cycle)
  - 3 MB L3 (on chip)
- 200 MHz 128-bit Bus
- SpecFP: 1356
- SpecInt: 810
- 44-bit addresses (18TB)
- 221M transistors
- 19.5 x 21.6 mm
- http://cpus.hp.com/technical_references/

Comments on Itanium

- Remarkably, the Itanium has many of the features more commonly associated with the dynamically-scheduled pipelines
  - strong emphasis on branch prediction, register renaming, scoreboarding, a deep pipeline with many stages before execution (to handle instruction alignment, renaming, etc.), and several stages following execution to handle exception detection
- Surprising that an approach whose goal is to rely on compiler technology and simpler HW seems to be at least as complex as dynamically scheduled processors!

EPIC/IA-64/Itanium principles

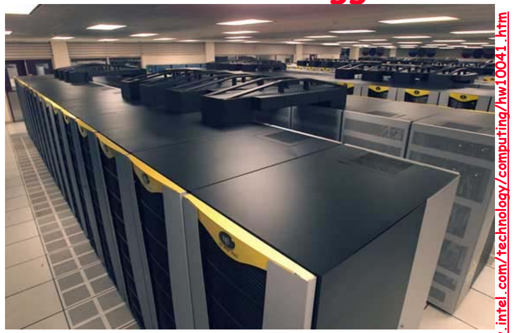
- Start loads early
  - advance loads - move above stores when alias analyis is incomplete
  - speculative loads - move above branches
- Predication to eliminate many conditional branches
  - 64 predicate registers
  - almost every instruction is predicated
- register rich
  - 128 integer registers (64 bits each)
  - 128 floating-point registers
- Independence architecture
  - VLIW flavor, but fully interlocked (i.e., no delay slots)
  - three 41-bit instruction syllables per 128-bit "bundle"
  - each bundle contains 5 "template bits" which specify independence of following syllables (within bundle and between bundles)
- unbundled branch architecture
  - eight branch registers
  - multiway branches
- Rotating register files
  - lower 48 of the predicate registers rotate
  - lower 96 of the integer registers rotate

Itanium Timeline

- 1981: Bob Rau leads Polycyclic Architecture project at TRW/ESL
- 1983: Josh Fisher describes ELI-512 VLIW design and trace scheduling
- 1983-1988: Rau at Cydrome works on VLIW design called Cydra-5, but company folds 1988
- 1984-1990: Fisher at Multiflow works on VLIW design called Trace, but company folds 1990
- 1988: Dick Lampman at HP hires Bob Rau and Mike Schlansker from Cydrome and also gets IP rights from Cydrome
- 1989: Rau & Schlansker begin FAST (Fine-grained Architecture & Software Technologies) research project at HP; later develop HP PlayDoh architecture
- 1990-1993: Bill Worley leads PA-WW (Precision Architecture Wide-Word) effort at HP Labs to be successor to PA-RISC architecture; also called SP-PA (Super-Parallel Processor Architecture) & SWS (SuperWorkStation)
- HP hires Josh Fisher, input to PA-WW
- Input to PA-WW from Hitachi team, led by Yasuyuki Okada
- 1991: Hans Mulder joins Intel to start work on a 64-bit architecture
- 1992: Worley recommends HP seek a semiconductor manufacturing partner
- 1993: HP starts effort to develop PA-WW as a product
- Dec 1993: HP investigates partnership with Intel
- June 1994: announcement of cooperation between HP & Intel; PA-WW starting point for joint design; John Crawford of Intel leads joint team
- 1997: the term EPIC is coined
- Oct 1997: Microprocessor Forum presentations by Intel and HP
- July 1998: Carole Dulong of Intel, "The IA-64 Architecture at Work," IEEE Computer
- Feb 1999: release of ISA details of IA-64
- 2001: Intel marketing prefers IPF (Itanium Processor Family) to IA-64
- May 2001 - Itanium (Merced)
- July 2002 - Itanium 2 (McKinley)
- Aug 2004: "Itanium sales fall $13.4bn shy of $14bn forecast" (The Register)
- Dec 2004: HP transfers last of Itanium development to Intel

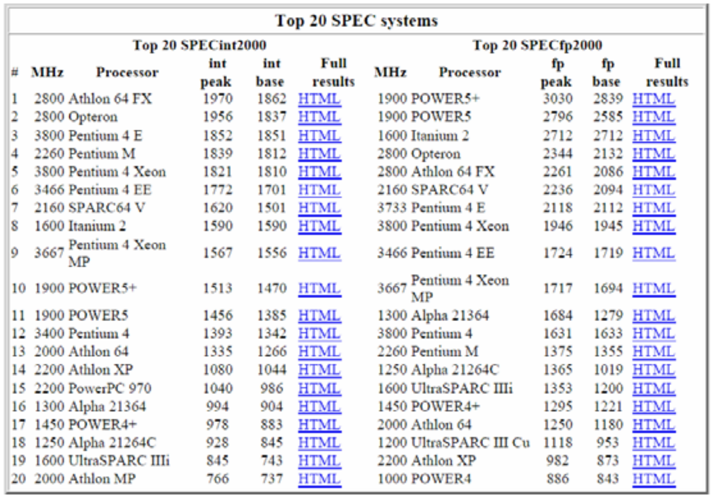
(based on http://www.cs.clemson.edu/~mark/epic.html)

Itanium – rumours exaggerated?



http://www.intel.com/technology/computing/hw10041.htm

- NASA's 10,240-processor Columbia supercomputer is built from 20 Altix systems, each powered by 512 Intel Itanium 2 processors. Peak performance 42.7 TeraFlops. Runs Linux. (Image courtesy of Silicon Graphics, Inc.)
- SGI has similar contracts at
  - Japan Atomic Energy Research Institute (JAERI) (2048 processors eventually)
  - Leibniz Rechenzentrum Computing Center (LRZ) at the Bavarian Academy of Sciences and Humanities, Munich (3328 processors eventually)



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	2800	Athlon 64 FX	1970	1862	HTML	1900	POWER5+	3030	2839	HTML
2	2800	Opteron	1956	1837	HTML	1900	POWER5	2796	2585	HTML
3	3800	Pentium 4 E	1852	1851	HTML	1600	Itanium 2	2712	2712	HTML
4	2260	Pentium M	1839	1812	HTML	2800	Opteron	2344	2132	HTML
5	3800	Pentium 4 Xeon	1821	1810	HTML	2800	Athlon 64 FX	2261	2086	HTML
6	3466	Pentium 4 EE	1772	1701	HTML	2160	SPARC64 V	2236	2094	HTML
7	2160	SPARC64 V	1620	1501	HTML	3733	Pentium 4 E	2118	2112	HTML
8	1600	Itanium 2	1590	1590	HTML	3800	Pentium 4 Xeon	1946	1945	HTML
9	3667	Pentium 4 Xeon MP	1567	1556	HTML	3466	Pentium 4 EE	1724	1719	HTML
10	1900	POWER5+	1513	1470	HTML	3667	Pentium 4 Xeon MP	1717	1694	HTML
11	1900	POWER5	1456	1385	HTML	1300	Alpha 21364	1684	1279	HTML
12	3400	Pentium 4	1393	1342	HTML	3800	Pentium 4	1631	1633	HTML
13	2000	Athlon 64	1335	1266	HTML	2260	Pentium M	1375	1355	HTML
14	2200	Athlon XP	1080	1044	HTML	1250	Alpha 21264C	1365	1019	HTML
15	2200	PowerPC 970	1040	986	HTML	1600	UltraSPARC IIIi	1353	1200	HTML
16	1300	Alpha 21364	994	904	HTML	1450	POWER4+	1295	1221	HTML
17	1450	POWER4+	978	883	HTML	2000	Athlon 64	1250	1180	HTML
18	1250	Alpha 21264C	928	845	HTML	1200	UltraSPARC III Cu	1118	953	HTML
19	1600	UltraSPARC IIIi	845	743	HTML	2200	Athlon XP	982	873	HTML
20	2000	Athlon MP	766	737	HTML	1000	POWER4	886	843	HTML

Aces Hardware analysis of SPEC benchmark data
http://www.aceshardware.com/SPECmine/top.jsp

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?

Summary #3: Software Scheduling

- Instruction Level Parallelism (ILP) found either by compiler or hardware.
- Loop level parallelism is easiest to see
  - SW dependencies/compiler sophistication determine if compiler can unroll loops
  - Memory dependencies hardest to determine => Memory disambiguation
  - Very sophisticated transformations available
- Trace Scheduling to Parallelize If statements
- Superscalar and VLIW: CPI < 1 (IPC > 1)
  - Dynamic issue vs. Static issue
  - More instructions issue at same time => larger hazard penalty
  - Limitation is often number of instructions that you can successfully fetch and decode per cycle