Graduate Computer Architecture

Chapter 4 – Explore Instruction Level Parallelism with Software Approaches

Basic Pipeline Scheduling

- Find sequences of unrelated instructions
- Compiler's ability to schedule
  - Amount of ILP available in the program
  - Latencies of the functional units
- Latency assumptions for the examples
  - Standard MIPS integer pipeline
  - No structural hazards (fully pipelined or duplicated units
  - Latencies of FP operations:

Instruction producing result	**Instruction using result**	**Latency**
FP ALU op	FP ALU op	3
FP ALU op	SD	2
LD	FP ALU op	1
LD	SD	0

Basic Scheduling

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

Sequential MIPS Assembly Code

```
Loop:   LD      F0, 0(R1)
        ADDD    F4, F0, F2
        SD      0(R1), F4
        SUBI    R1, R1, #8
        BNEZ    R1, Loop
```

Pipelined execution:

```
Loop:   LD      F0, 0(R1)       1
        stall                   2
        ADDD    F4, F0, F2      3
        stall                   4
        stall                   5
        SD      0(R1), F4       6
        SUBI    R1, R1, #8      7
        stall                   8
        BNEZ    R1, Loop        9
        stall                  10
```

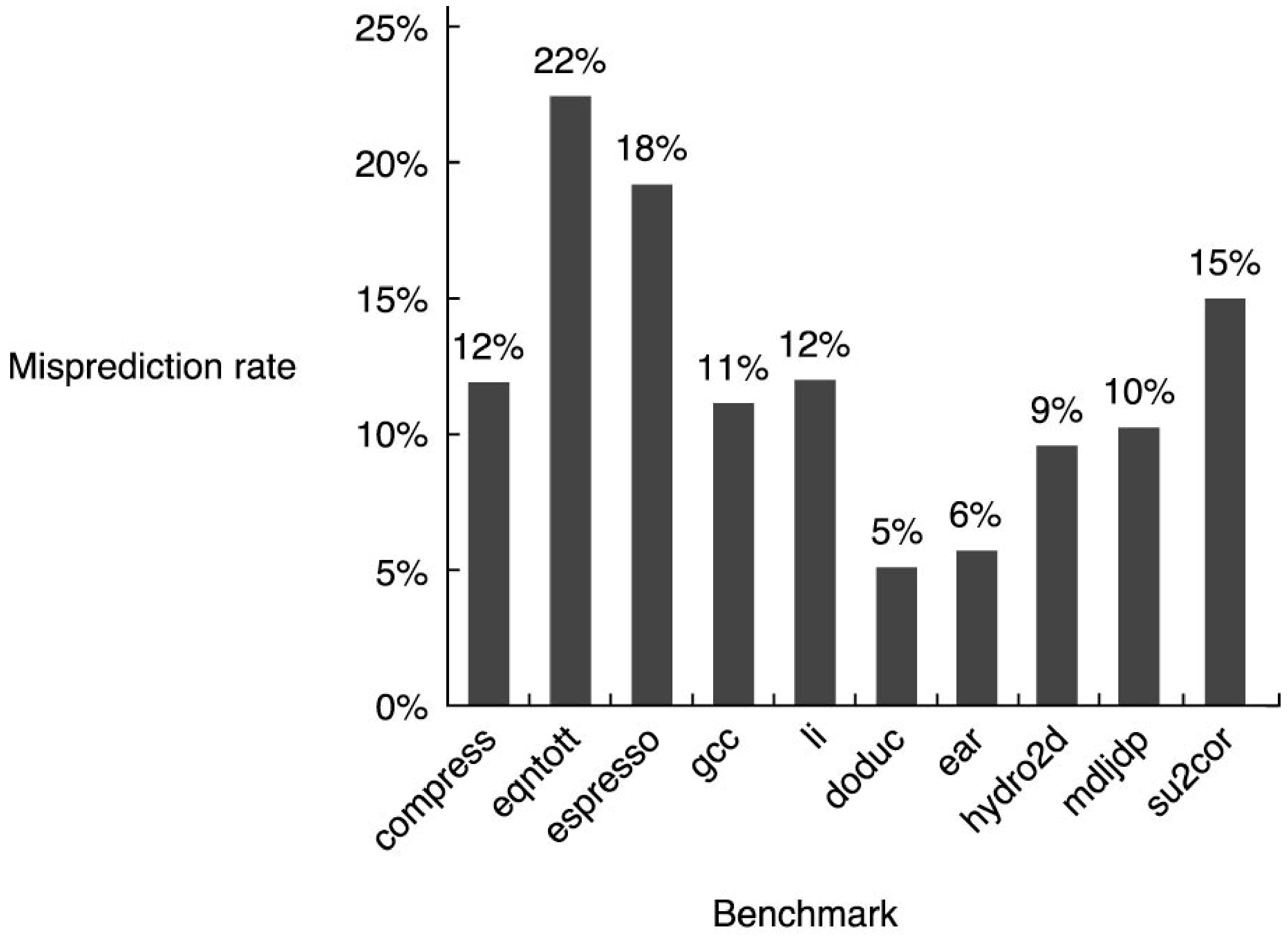
Scheduled pipelined execution:

```
Loop:   LD      F0, 0(R1)       1
        SUBI    R1, R1, #8      2
        ADDD    F4, F0, F2      3
        stall                   4
        BNEZ    R1, Loop        5
        SD      8(R1), F4       6
```

Static Branch Prediction: Using Compiler Technology

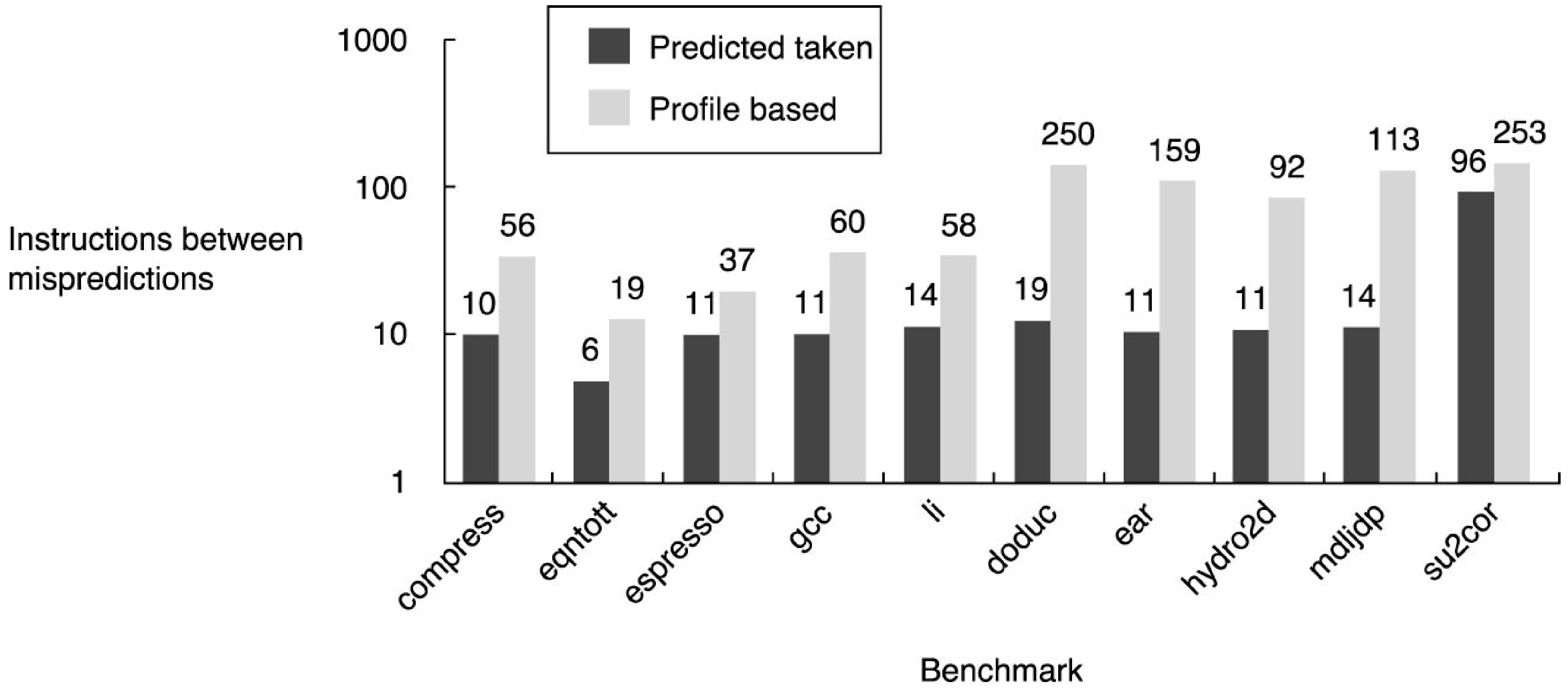
- How to statically predict branches?
  - Examination of program behavior
    - Always predict taken (on average, 67% are taken)
      - Mis-prediction rate varies large (9%—59%)
    - Predict backward branches taken, forward branches un-taken (mis-prediction rate < 30% -- 40%)
  - Profile-based predictor: use profile information collected from earlier runs

Mis-prediction Rate for a Profile-based Predictor



FP is better than integer

Prediction-taken VS. Profile-based Predictor



Average 20 for prediction-taken
110 for profile-based

Standard Deviations are large

Static Multiple Issue: The VLIW Approach

- VLIW (very long instruction word)
- Issue a fixed number of instructions formatted as…
  - Package multiple operation into one very long instruction
  - – or – A fixed instruction packet with the parallelism among instructions explicitly indicated by instruction
    - Also known as EPIC – explicitly parallel instruction computers
- Rely on compiler to…
  - Minimize the potential hazard stall
  - Actually format the instructions in a potential issue packet so that HW need not check explicitly for dependencies. Compiler ensures…
    - Dependences within the issue packet cannot be present
    - – or – Indicate when a dependence may occur

Compiler does most of the work of finding and scheduling instructions for parallel execution

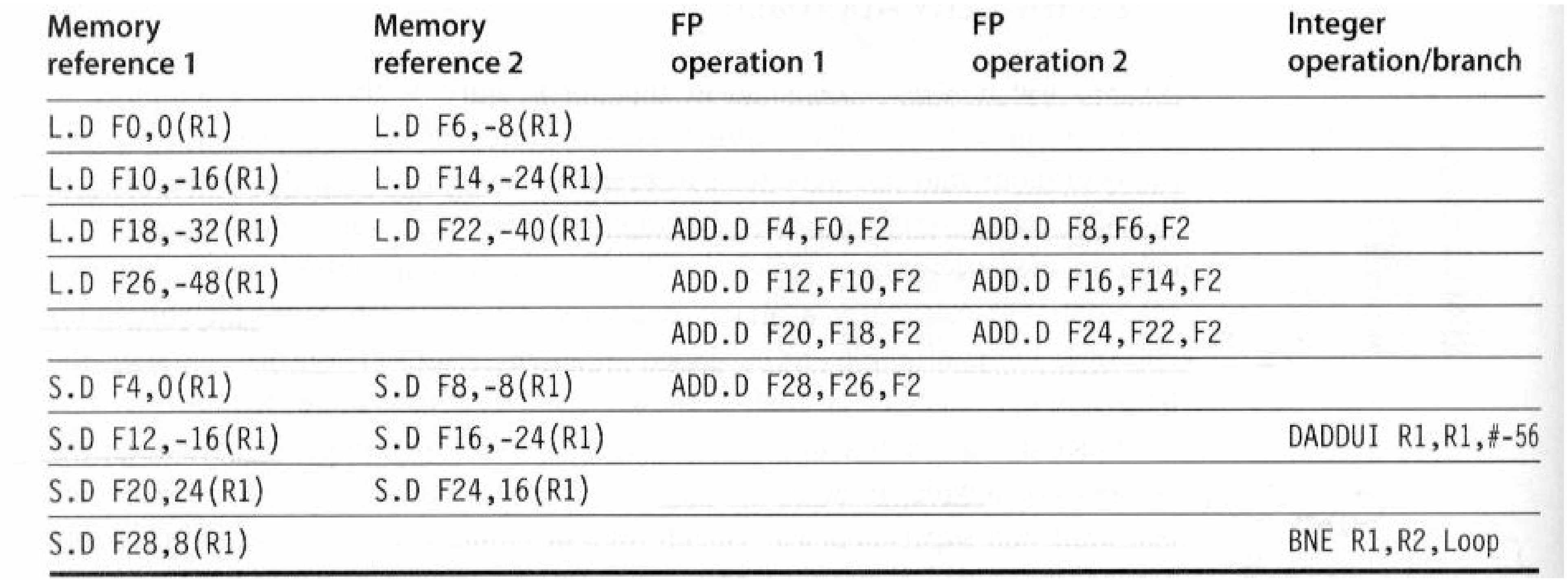
Basic VLIW

- A VLIW uses multiple, independent functional units
- A VLIW packages multiple independent operations into one very long instruction
  - The burden for choosing and packaging independent operations falls on compiler
  - HW in a superscalar makes these issue decisions is unneeded
    - This advantage increases as the maximum issue rate grows
- Here we consider a VLIW processor might have instructions that contain 5 operations, including 1 integer (or branch), 2 FP, and 2 memory references
  - Depend on the available FUs and frequency of operation

Basic VLIW (Cont.)

- VLIW depends on enough parallelism for keeping FUs busy
  - Loop unrolling and then code scheduling
  - Compiler may need to do local scheduling and global scheduling
    - Techniques to enhance LS and GS will be mentioned later
  - For now, assume we have a technique to generate long, straight-line code sequences

Loop Unrolling in VLIW



Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.29 clocks per iteration

23 ops in 9 clock, average 2.5 ops per clock,

60% FUs are used

Note: Need more registers in VLIW

VLIW Problems – Technical

- Increase in code size
  - Ambitious loop unrolling
  - Whenever instruction are not full, the unused FUs translate to waste bits in the instruction encoding
    - An instruction may need to be left completely empty if no operation can be scheduled
    - Clever encoding or compress/decompress

VLIW Problems – Logistical

- Synchronous VS. Independent FUs
  - Early VLIW – all FUs must be kept synchronized
    - A stall in any FU pipeline may cause the entire processor to stall
  - Recent VLIW – FUs operate more independently
    - Compiler is used to avoid hazards at issue time
    - Hardware checks allow for unsynchronized execution once instructions are issued.

VLIW Problems – Logistical

- Binary code compatibility
  - Code sequence makes use of both the instruction set definition and the detailed pipeline structure (FUs and latencies)
  - Need migration between successive implementations, or between implementations ➔ recompliation
  - Solution
    - Object-code translation or emulation
    - Temper the strictness of the approach so that binary compatibility is still feasible (IA-64)

Advantages of Superscalar over VLIW

- Old codes still run
  - Like those tools you have that came as binaries
  - HW detects whether the instruction pair is a legal dual issue pair
    - If not they are run sequentially
- Little impact on code density
  - Don’t need to fill all of the can’t issue here slots with NOP’s
- Compiler issues are very similar
  - Still need to do instruction scheduling anyway
  - Dynamic issue hardware is there so the compiler does not have to be too conservative

Advance Compiler Support for Exposing and Exploiting ILP

- Discuss compiler technology for increasing the amount of parallelism that we can exploit in a program
  - Loop unrolling
  - Detecting and Enhancing loop-level parallelism
  - Finding and eliminating dependent computations
  - Software pipelining: symbolic loop unrolling
  - Global code scheduling
    - Trace scheduling
    - Superblock

Review: Unrolled Loop that Minimizes Stalls for Scalar

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

```
1 Loop: LD     F0,0(R1)
2       LD     F6,-8(R1)
3       LD     F10,-16(R1)
4       LD     F14,-24(R1)
5       ADDD   F4,F0,F2
6       ADDD   F8,F6,F2
7       ADDD   F12,F10,F2
8       ADDD   F16,F14,F2
9       SD     0(R1),F4
10      SD     -8(R1),F8
11      SD     -16(R1),F12
12      SUBI   R1,R1,#32
13      BNEZ   R1,LOOP
14      SD     8(R1),F16    ; 8-32 = -24
```

14 clock cycles, or 3.5 per iteration

Detect and Enhance Loop Level Parallelism

- Loop-level parallelism: analyzed at the source or near level
  - Most ILP analysis: analyzed once instructions have been generated
- Loop-level analysis
  - Determine what dependences exist among the operands in a loop across the iterations of that loop
  - Determine whether data accesses in later iterations are dependent on data values produced in earlier iterations
    - Loop-carried dependence (LCD) VS. loop-level parallel
    - LCD forces successive loop iterations to execute in series
  - Finding loop-level parallelism involves recognizing structures such as loops, array references, and induction variable computations
    - The compiler can do this analysis more easily at or near the source level

Example 1

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

Assume A, B, and C are distinct, non-overlapping arrays

- S1 uses a value computed by S1 in an earlier iteration (A[i])
  - Loop-carried dependence
- S2 uses a value computed by S2 in an earlier iteration (B[i])
  - Loop-carried dependence
- S2 uses a value computed by S1 in the same iteration (A[i+1])
  - If not loop-carried
  - Multiple iterations can execute in parallelism, as long as dependent statements in an iteration are kept in order

Example 2

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

The existence of loop-carried dependence does not prevent parallelism

- S1 uses a value computed by S2 in an earlier iteration (B[i+1])
  - Loop-carried dependence
- Dependence is not circular
  - Neither statement depends on itself, and although S1 depends on S2, S2 does not depend on S1
- A loop is parallel if it can be written without a cycle in the dependences
  - Absence of a cycle give a partial ordering on the statements

Example 2 (Cont.)

```
A[1] = A[1] + B[1]
for (i=1; i <= 99; i=i+1) {
   B[i+1] = C[i] + D[i];
   A[i+1] = A[i+1] + B[i+1];
 }
B[101] = C[100] + D[100]
```

- Transform the code in the previous slide to conform to the partial ordering and expose the parallelism
- No longer loop-carried. Iterations of the loop may be overlapped, provided the statements in each iteration are kept in order

Detect and Eliminate Dependences

- Trick is to find and remove dependencies
  - Simple for single variable accesses
  - More complex for pointers, array references, etc.
- Things get easier if
  - Non-cyclic dependencies – No loop-carried dependencies
  - Recurrent dependency distance is larger (more ILP can be explored)
    - Recurrence: when a variable is defined based on the value of that variable in an earlier iteration
    - ex. Y[i] = Y[i-5] + Y[i] (dependence distance = 5)
  - Array index calculation is consistent
    - Affine array indices ai+b where a and b are constants
    - GCD test (pp. 324) to determine two affine functions can have the same value for different indices between the loop bound

Detect and Eliminate Dependences -- Difficulties

- Barriers to analysis
  - Reference via pointers rather than predictable array indices
  - Array indexing is Indirect through another array: x[y[i]] (non-affine)
    - Common sparse array accesses
  - False dependency: for some input values a dependence may exist
    - Run time checks must be used to determine the dependent case
- General: NP hard problem
  - Specific cases can be done precisely
  - Current problem: a lot of special cases that don’t apply often
  - The good general heuristic is the holy grail
  - Points-to analysis: analyzing programs with pointers (pp. 326—327)

Eliminating Dependent Computations – Techniques

- Eliminate or reduce a dependent computation by back substitution – within a basic block and within loop
- Algebraic simplification + Copy propagation (eliminate operations that copy values within a basic block)
  - Reduce multiple increment of array indices during loop unrolling and move increments across memory addresses in Section 4.1

```
DADDUI   R1, R2, #4
DADDUI   R1, R1, #4
```

$\Rightarrow$

```
DADDUI   R1, R2, #8
```

- Tree-height reduction – increase the code parallelism

```
DADDUI   R1, R2, R3
DADDUI   R4, R1, R6
DADDUI   R8, R4, R7
```

$\Rightarrow$

```
DADDUI   R1, R2, R3
DADDUI   R4, R6, R7
DADDUI   R8, R1, R4
```

Eliminating Dependent Computations – Techniques (Cont.)

- Optimization related to recurrence
  - Recurrence: expressions whose value on one iteration is given by a function that depends on previous iteration
  - When a loop with a recurrence is unrolled, we may be able to algebraically optimized the unrolled loop, so that the recurrence need only be evaluated once per unrolled iteration
    - sum = sum + x ➔ sum = sum + x1 + x2 + x3 + x4 + x5 (5 dependent operations) ➔ sum = ((sum + x1) + (x2 + x3)) + (x4 + x5) (3 dependent operations)

An Example to Eliminate False Dependences

```
for (i=1; i <= 100; i=i+1) {
  Y[i] = X[i] / C;   /* S1 */
  X[i] = X[i] + C;  /* S2 */
  Z[i] = Y[i] + C;  /* S3 */
  Y[i] = C – Y[i];  /* S4 */
}
```

True dependence: S1→S3, S1→S4 (Y[i])
Antidependence: S1→S2 (X[i])
Antidependence: S3→S4 (Y[i])
Output dependence: S1→S4(Y[i])

```
for (i=1; i <= 100; i=i+1) {
  T[i] = X[i] / C;   /* S1 */
  X1[i] = X[i] + C;  /* S2 */
  Z[i] = T[i] + C;  /* S3 */
  Y[i] = C – T[i];  /* S4 */
}
```

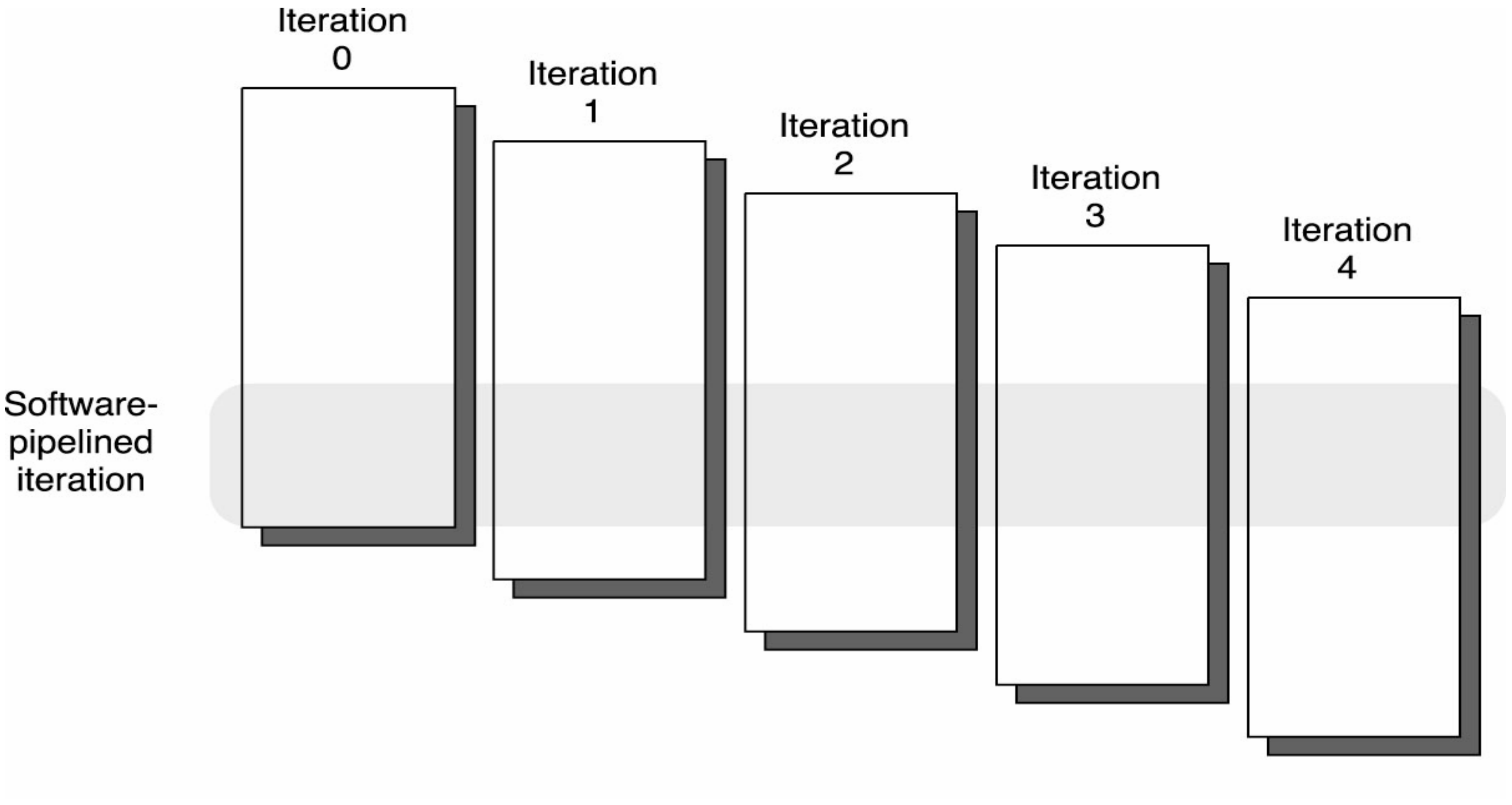
Antidependence: X→X1
Antidependence: Y→T
Output dependence: Y→T

X has been renamed to X1 →
Compiler replace X by X1 -- or –
Copy X1 to X

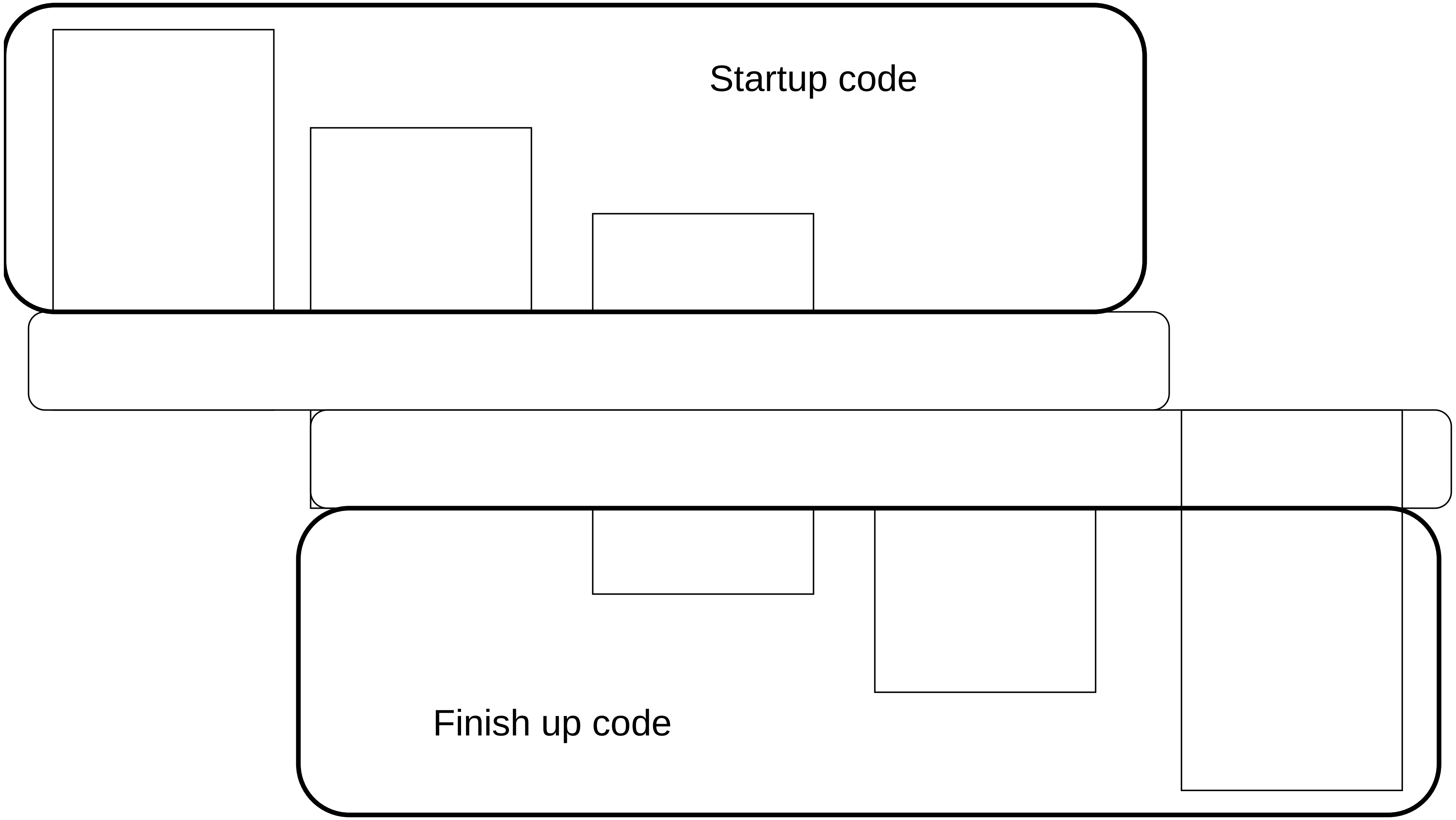
Software Pipelining

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from <u>different</u> iterations
- Software pipelining (Symbolic loop unrolling): reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop
- Idea is to separate dependencies in original loop body
  - Register management can be tricky but idea is to turn the code into a single loop body
  - In practice both unrolling and software pipelining will be necessary due to the register limitations

Software Pipelining (Cont.)



Software Pipelining (Cont.)



Software Pipelining Example

Before: Unrolled 3 times

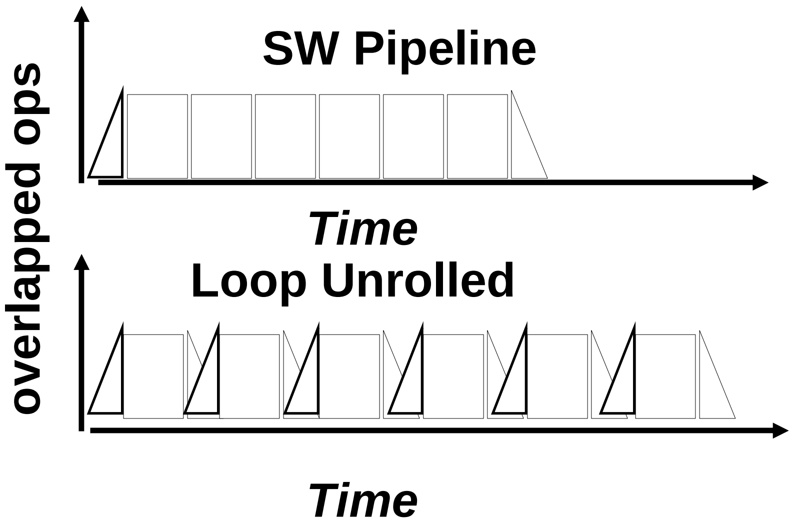
```
1  LD   F0,0(R1)
2  ADDD F4,F0,F2
3  SD   0(R1),F4
4  LD   F6,-8(R1)
5  ADDD F8,F6,F2
6  SD   -8(R1),F8

7  LD   F10,-16(R1)
8  ADDD F12,F10,F2
9  SD   -16(R1),F12
10 SUBI R1,R1,#24
11 BNEZ R1,LOOP
```

After: Software Pipelined

```
1  SD   0(R1),F4  ; Stores M[i]
2  ADDD F4,F0,F2  ; Adds to M[i-1]
3  LD   F0,-16(R1); Loads M[i-2]
4  SUBI R1,R1,#8
5  BNEZ R1,LOOP
```

Software Pipelining (Cont.)

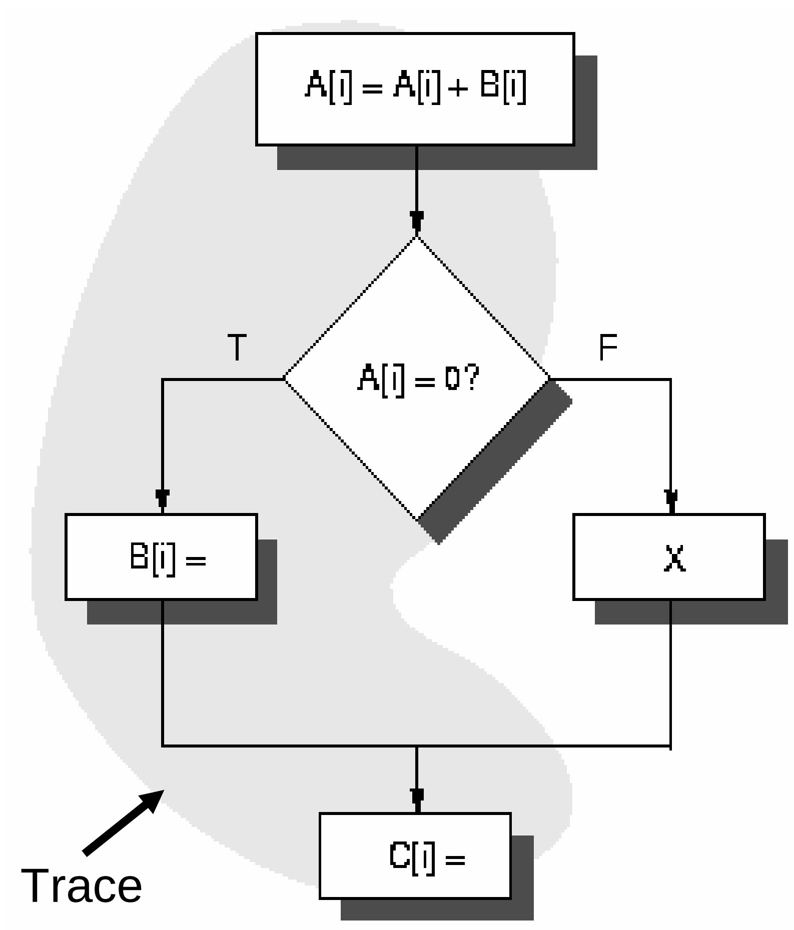
- Symbolic Loop Unrolling
- Maximize result-use distance
- Less code space than unrolling
- Fill & drain pipe only once per loop vs. once per each unrolled iteration in loop unrolling



Global Code Scheduling

- Aim to compact a code fragment with internal control structure into the shortest possible sequence that preserves the data and control dependences
  - Reduce the effect of control dependences arising from conditional non-loop branches by moving code
    - Loop branches can be solved by unrolling
- Require estimates of the relative frequency of different paths (Then…Else)
- Global code scheduling example (next slide)
  - Suppose the shaded part (Then) is more frequently executed than the ELSE part
- In general, extremely complex

Global Code Scheduling Example



```
LD        R4, 0(R1)           ; load A
LD        R5, 0(R2)           ; load B
DADDUR4, R4, R5               ; Add to A
SD                  R4, 0(R1)           ; store
   A
…
BNEZ      R4, elsepart        ; Test A
…
SD                  …, 0(R2)
          ; Store B
…
J         join                ; j over else
elsepart:…                    ;
          X                   ; code for X
          …
join:     …                   ; after if SW
          …, 0(R3)            ; store C[i]
```

Global Code Scheduling Example (Cont.)

- Try to move the assignment of B and C before the test of A
- Move B
  - If an instruction in X depends on the original value of B, moving B before BNEZ will influence the data dependence
  - Make a shadow copy of B before IF and use that shadow copy in X?
    - Complex to implement
    - Slow down the program if the trace selected is not optimal
      - Require additional instructions to execute
- Move C
  - To the THEN part → A copy of C must be put in the ELSE part
  - Before the BNEZ → If can be moved → the copy of C in the ELSE part can be eliminated

Factors in Moving B

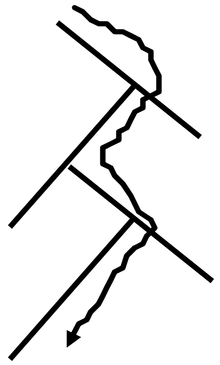
- The compiler will consider the following factors
  - Relative execution frequencies of THEN and ELSE
  - Cost of executing the computing and assignment to B above branch
    - Any empty instruction issue slots and stalls above branch?
  - How will the movement of B change the execution time for THEN
  - Is B the best code fragment that can be moved? How about C or others?
  - The cost of the compensation code that may be necessary for ELSE

Trace Scheduling

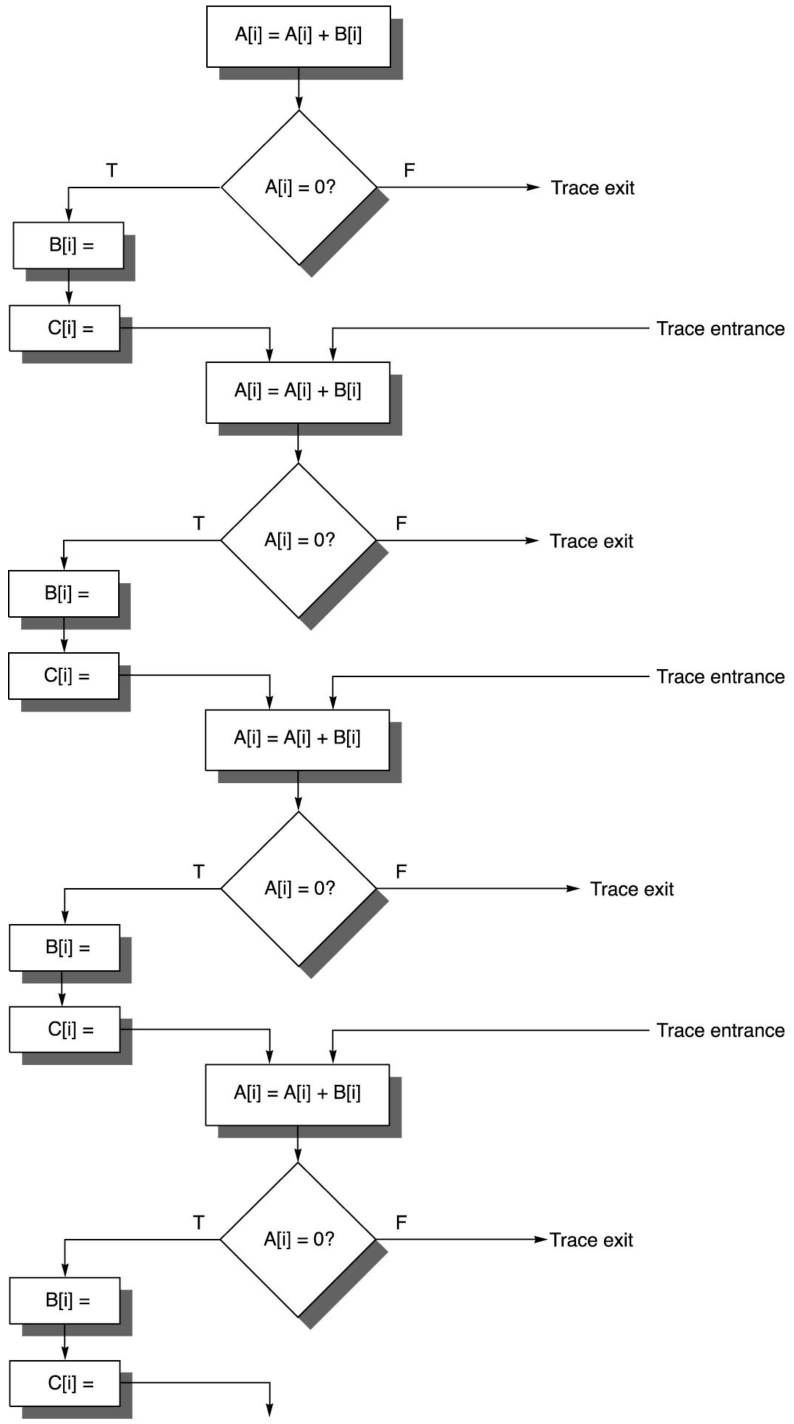
- Useful for processors with a large number of issues per clock, where conditional or predicted execution (Section 4.5) is inappropriate or unsupported, and where loop unrolling may not be sufficient by itself to uncover enough ILP
- A way to organize the global code motion process, so as to simplify the code scheduling by incurring the costs of possible code motion on the less frequent paths
- Best used where profile information indicates significant differences in frequency between different paths and where the profile information is highly indicative of program behavior independent of the input

Trace Scheduling (Cont.)

- Parallelism across conditional branches other than loop branches
- Looking for the critical path across conditional branches
- Two steps:
  - Trace Selection: Find likely sequence of multiple basic blocks (trace) of long sequence of straight-line code
  - Trace Compaction
    - Squeeze trace into few VLIW instructions
    - Move trace before the branch decision
    - Need bookkeeping code in case prediction is wrong



- Compiler undoes bad guess (discards values in registers)
- Subtle compiler bugs mean wrong answer vs. poor performance; no hardware interlocks



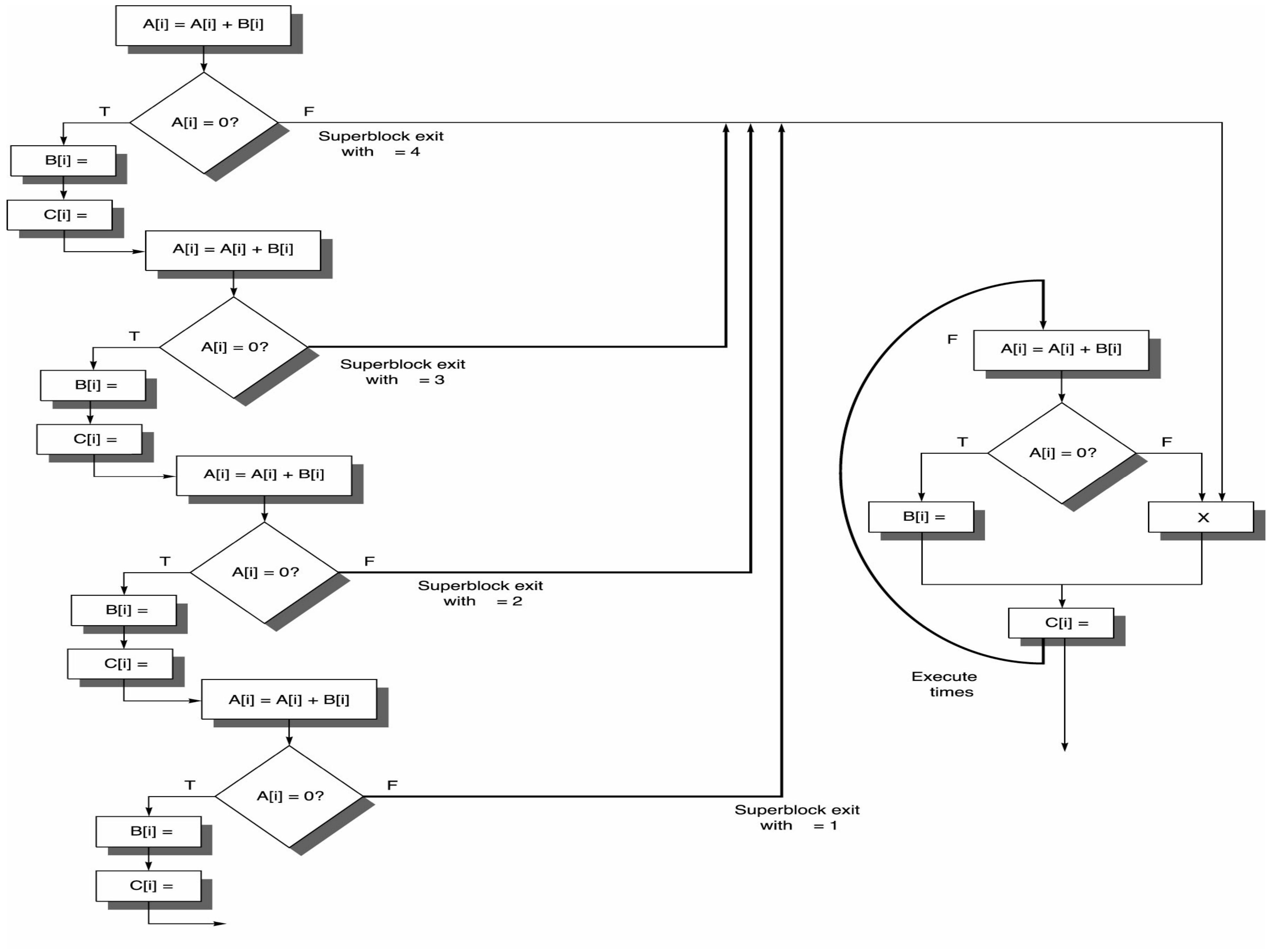
This trace is obtained by assuming that the program fragment in Figure 4.8 is the inner loop and unwinding it four times, treating the shaded portion in Figure 4.8 as the likely path.

Trace Scheduling (Cont.)

- Simplify the decisions concerning global code motion
- Branches are viewed as jumps into or out of the selected trace (the most probable path)
- Additional book-keeping code will often be needed on the entry of exit point
  - If the trace is so much more probable than the alternatives that the cost of the bookkeeping code need not be a deciding factor
- Trace scheduling is good for scientific code
  - Intensive loops and accurate profile data
- But unclear if this approach is suitable for programs that are less simply characterized and less loop-intensive

Superblocks

- Drawback of trace scheduling: the entries and exits into the middle of the trace cause significant complications
  - Compensation code, and hard to assess their cost
- Superblocks – similar to trace, but
  - Single entry point but allow multiple exits
- In a loop that has a single loop exit based on a count, the resulting superblocks have only one exit
- Use tail duplication to create a separate block corresponding to the portion of the trace after the entry


A[i] = A[i] + B[i]
A[i] = 0?
T
F
B[i] =
C[i] =
Superblock exit with = 4
Superblock exit with = 3
Superblock exit with = 2
Superblock exit with = 1
X
Execute times

Some Things to Notice

SW pipelining, loop unrolling, trace scheduling, superblocks

- Useful when the branch behavior is fairly predictable at compiler time
- Not totally independent techniques
  - All try to avoid dependence induced stalls
- Primary focus
  - Unrolling: reduce loop overhead of index modification and branch
  - SW pipelining: reduce single body dependence stalls
  - Trace scheduling/superblocks: reduce impact of branch walls
- Most advanced compilers attempt all
  - Result is a hybrid which blurs the differences
  - Lots of special case analysis changes the hybrid mix
- All tend to fail if branch prediction is unreliable

Conditional or Predicated Instructions

```
BNEZ R1, L
MOV R2, R3      ⇨   CMOVZ R2,R3, R1
L:
```

- Most common form is move
- Other variants
  - Conditional loads and stores
  - ALPHA, MIPS, SPARC, PowerPC, and P6 all have simple conditional moves
  - IA_64 supports full predication for all instructions
- Effect is to eliminating simple branches
  - Moves dependence resolution point from early in the pipe (branch resolution) to late in the pipe (register write) → forwarding…is more possible
  - Also changes a control dependence into a data dependen
  - Example if (B<0) A=-B; else A=B; -> (A=B); conditional MOV (A=-B)
- Net win since in global scheduling the control dependence is the key limiting complexity

Conditional Instruction in SuperScalar

First slot(Mem)	Second slot (ALU)
LW R1,40(R2)	ADD R3, R4, R5
	ADD R6, R3, R7
BEQZ R10, L	
LW R8, 20(R10)	
LW R9, 0(R8)	

First slot(Mem)	Second slot (ALU)
LW R1,40(R2)	ADD R3, R4, R5
LWC R8,20(R10),R10	ADD R6, R3, R7
BEQZ R10, L	
LW R9, 0(R8)	

- Waste a memory operation slot in the 2nd cycle
- data dependence stall if not taken

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 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 Can Chance Instruction Set?

- Superscalar processors decide on the fly how many instructions to issue
  - High HW complexity
- 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

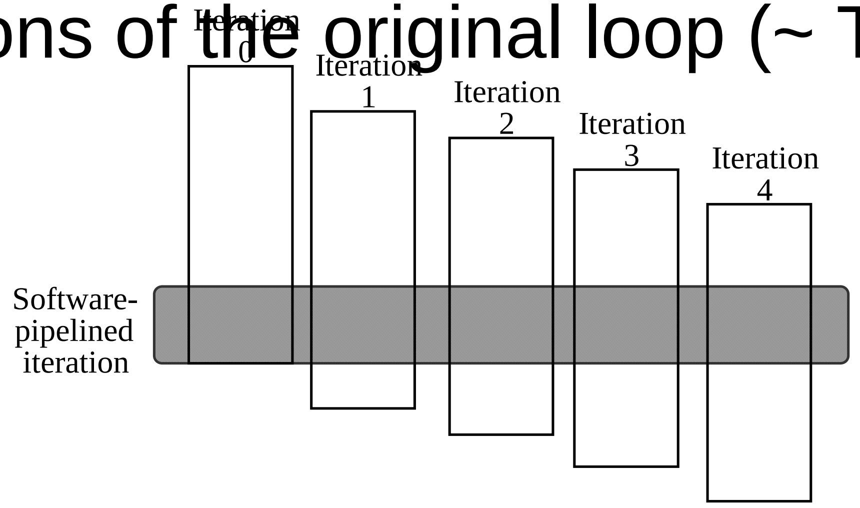
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

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

Recall: Software Pipelining

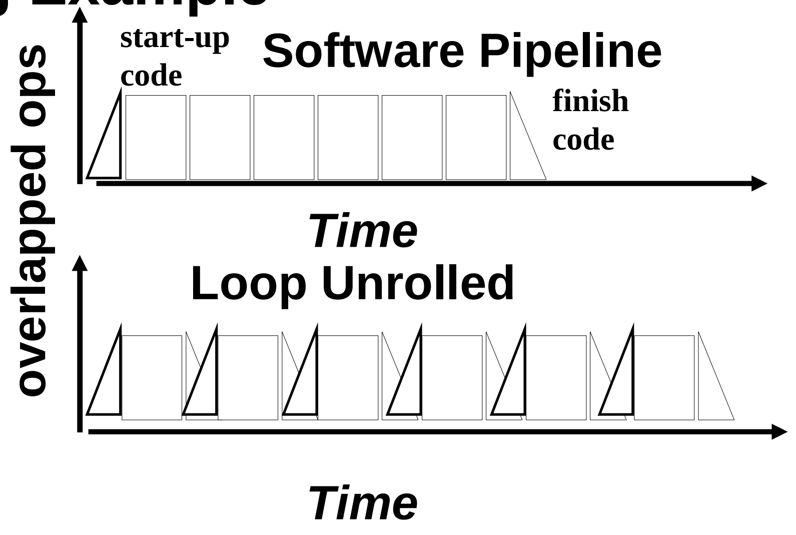
- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from <u>different</u> iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (~ Tomasulo in SW)



Software Pipelining Example

Show a software-pipelined version of the code:

```
Loop:   L.D      F0,0(R1)
        ADD.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDUI   R1,R1,#-8
        BNE      R1,R2,LOOP
```



Before: Unrolled 3 times

```
1   L.D      F0,0(R1)
2   ADD.D    F4,F0,F2
3   S.D      F4,0(R1)
4   L.D      F0,-8(R1)
5   ADD.D    F4,F0,F2
6   S.D      F4,-8(R1)
7   L.D      F0,-16(R1)
8   ADD.D    F4,F0,F2
9   S.D      F4,-16(R1)
10  DADDUI    R1,R1,#-24
11  BNE      R1,R2,LOOP
```

After: Software Pipelined

```
     L.D      F0,0(R1)                        start-up
     ADD.D    F4,F0,F2                        code
     L.D      F0,-8(R1)
1    S.D      F4,0(R1)   ;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    DADDUI   R1,R1,#-8
5    BNE      R1,R2,LOOP
     S.D      F4, 0(R1)                       finish
     ADDD     F4,F0,F2                        code
     S.D      F4,-8(R1)
```

2 fewer loop iterations

Software Pipelining with Loop Unrolling in VLIW

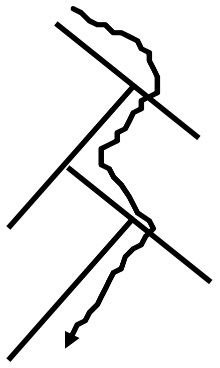
Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
L.D F0,-48(R1)	ST 0(R1),F4	ADD.D F4,F0,F2			1
L.D F6,-56(R1)	ST -8(R1),F8	ADD.D F8,F6,F2		DSUBUI R1,R1,#24	2
L.D F10,-40(R1)	ST -16(R1),F12	ADD.D F12,F10,F2		BNEZ R1,LOOP	3

- 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 prediction function units, but caches hard 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)"

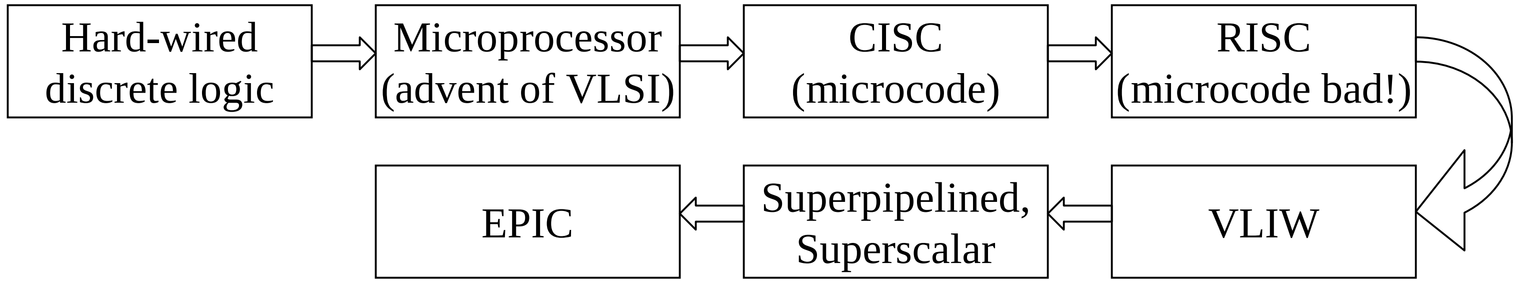
- IA-64: instruction set architecture; EPIC is type
  - EPIC = 2nd generation VLIW?
- Itanium™ is name of first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 μ process
- 128 64-bit integer registers + 128 82-bit floating 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?

What is EPIC?

- Explicitly Parallel Instruction-set Computing – a new hardware architecture paradigm that is the successor to VLIW (Very Long Instruction Word).
- EPIC features:
  - Parallel instruction encoding
  - Flexible instruction grouping
  - Large register file
  - Predicated instruction set
  - Control and data speculation
  - Compiler control of memory hierarchy

History

- A timeline of architectures:


Hard-wired discrete logic
Microprocessor (advent of VLSI)
CISC (microcode)
RISC (microcode bad!)
VLIW
Superpipelined, Superscalar
EPIC

- Hennessy and Patterson were early RISC proponents.
- VLIW (very long instruction word) architectures, 1981.
  - Alan Charlesworth (Floating Point Systems)
  - Josh Fisher (Yale / Multiflow – Multiflow Trace machines)
  - Bob Rau (TRW / Cydrome – Cydra-5 machine)

History (cont.)

- Superscalar processors
  - Tilak Agerwala and John Cocke of IBM coined term
  - First microprocessor (Intel i960CA) and workstation (IBM RS/6000) introduced in 1989.
  - Intel Pentium in 1993.
- Foundations of EPIC - HP
  - HP FAST (Fine-grained Architecture and Software Technologies) research project in 1989 included Bob Rau; this work later developed into the PlayDoh architecture.
  - In 1990 Bill Worley started the PA-Wide Word project (PA-WW). Josh Fisher, also hired by HP, made contributions to these projects.
  - HP teamed with Intel in 1994 – announced EPIC, IA-64 architecture in 1997.
  - First implementation was Merced / Itanium, announced in 1999.

IA-64 / Itanium

- IA-64 is not a complete architecture – it is more a set of high-level requirements for an architecture and instruction set, co-developed by HP and Intel.
- Itanium is Intel’s implementation of IA-64. HP announced an IA-64 successor to their PA-RISC line, but apparently abandoned it in favor of joint development of the Itanium 2.
- Architectural features:
  - No complex out-of-order logic in processor – leave it for compiler
  - Provide large register file, multiple execution units (again, compiler has visibility to explicitly manage)
  - Add hardware support for advanced ILP optimization: predication, control and data speculation, register rotation, multi-way branches, parallel compares, visible memory hierarchy
  - Load / store architecture – no direct-to-memory instructions

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 predict registers

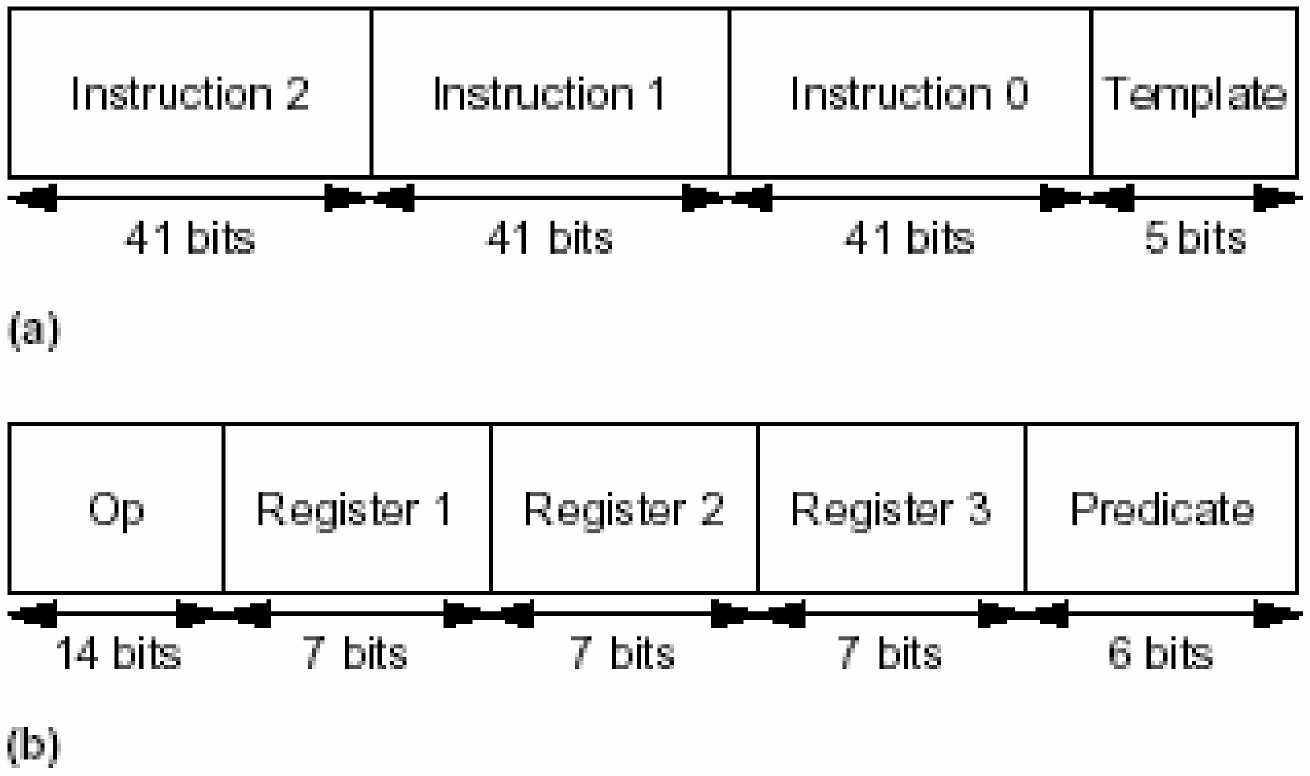
IA-64 Registers

- Both the integer and floating point registers support register rotation for registers 32-128.
- Register rotation is designed to ease the task of allocating of registers 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

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

Instruction Format



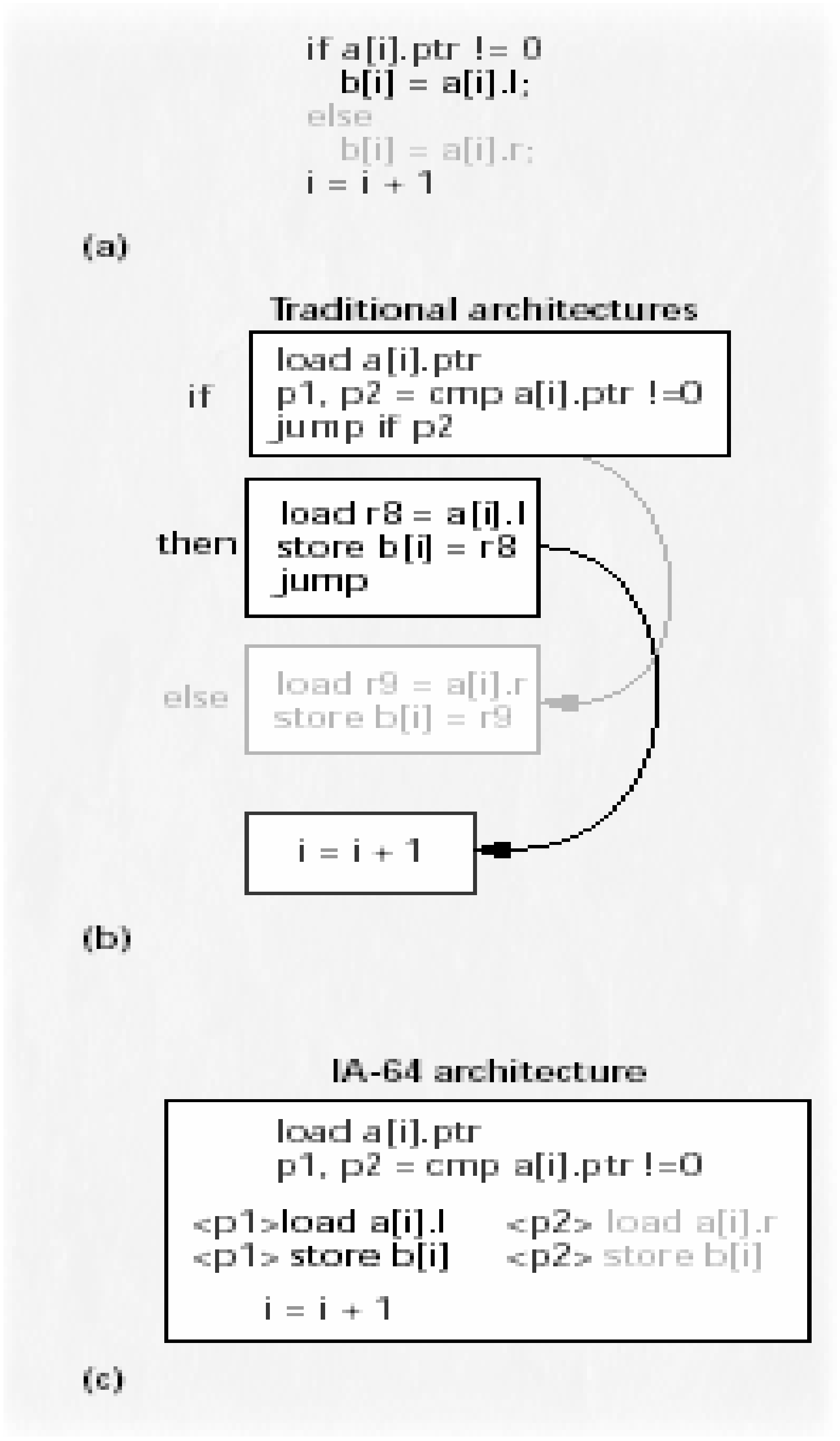
5 Types of Execution in Bundle

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

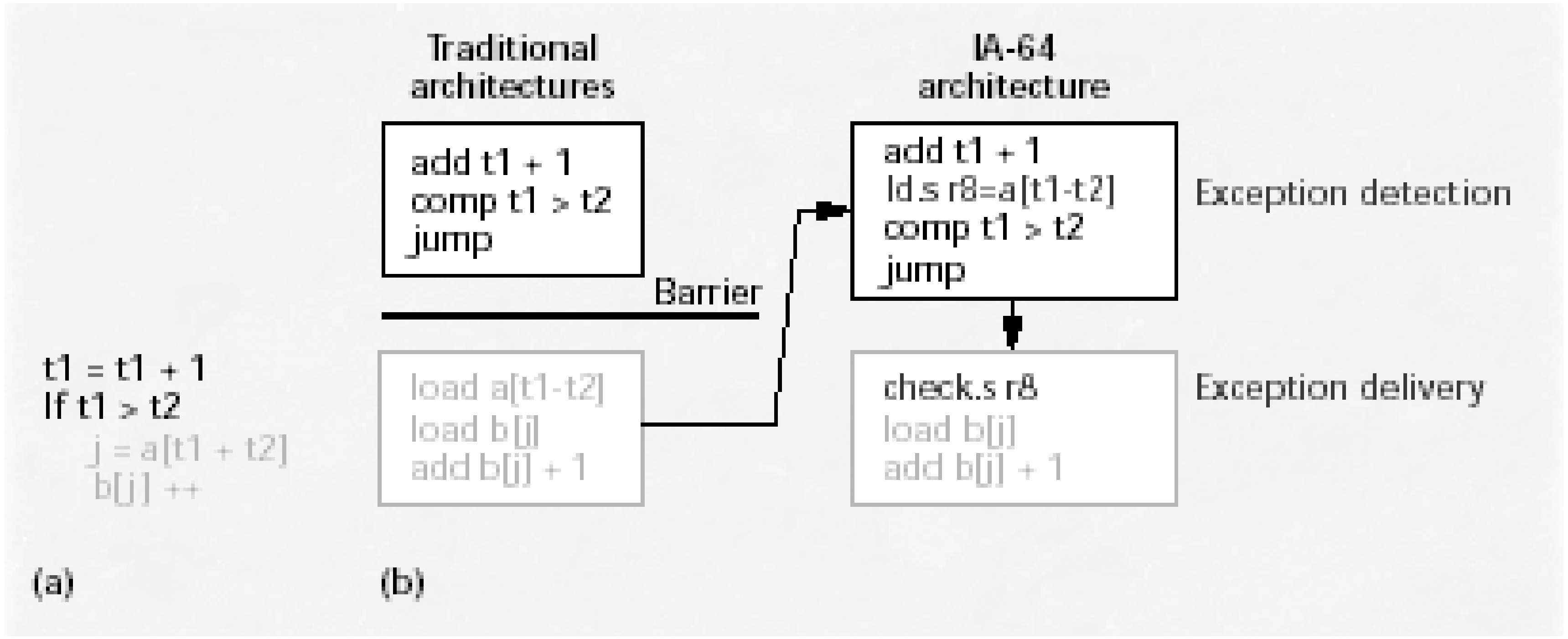
- **5-bit template field within each bundle describes both the presence of any stops associated with the bundle *and* the execution unit type required by each instruction within the bundle (see Fig 4.12 page 271)**

Template	Slot 0	Slot 1	Slot 2
0	M	I	I
1	M	I	I
2	M	I	I
3	M	I	I
4	M	L	X
5	M	L	X
8	M	M	I
9	M	M	I
10	M	M	I
11	M	M	I
12	M	F	I
13	M	F	I
14	M	M	F
15	M	M	F
16	M	I	B
17	M	I	B
18	M	B	B
19	M	B	B
22	B	B	B
23	B	B	B
24	M	M	B
25	M	M	B
28	M	F	B
29	M	F	B

- **Predicated Execution**



Speculative Loads



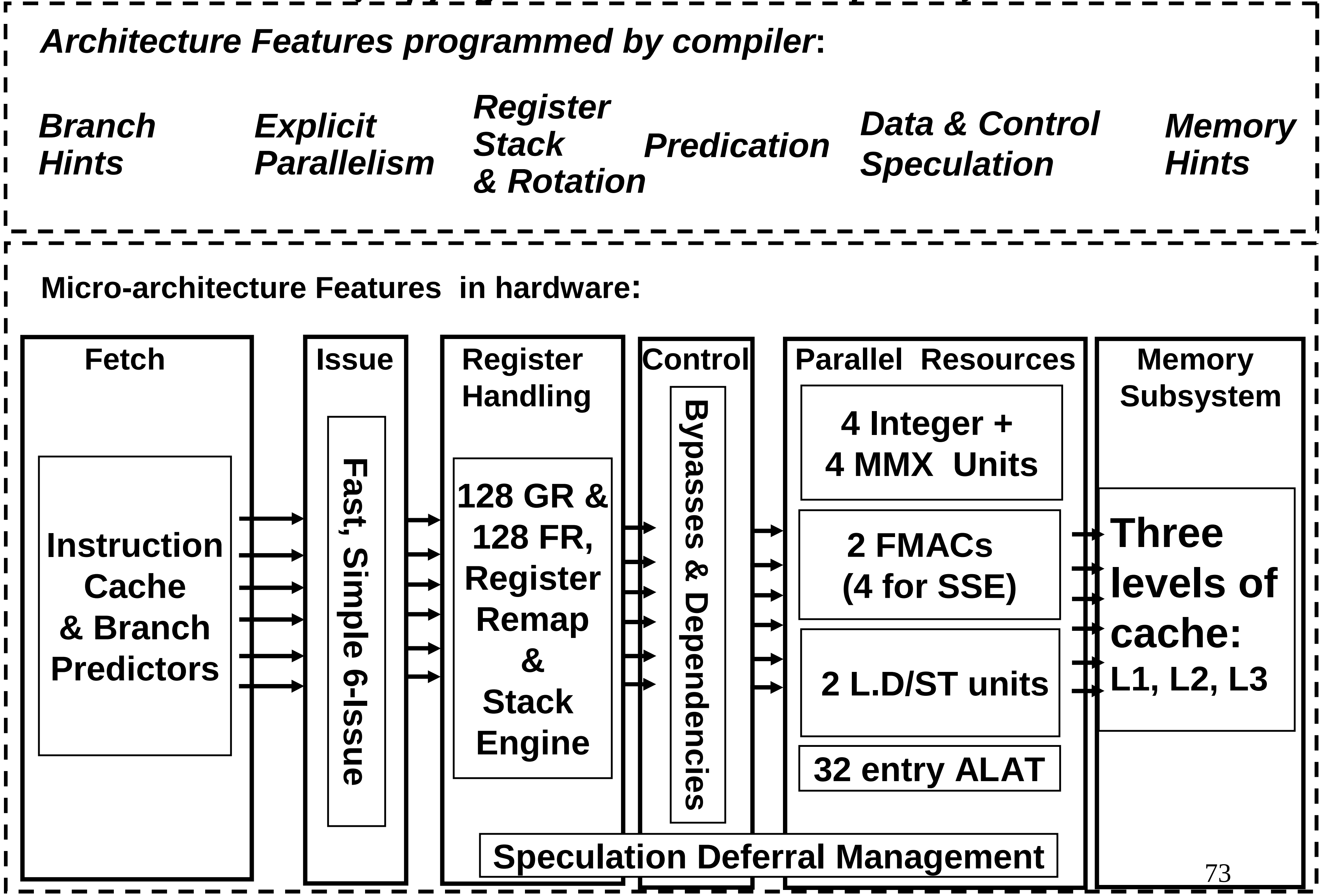
Itanium™ Machine Characteristics

(Copyright: Intel at Hotchips '00)

Frequency	800 MHz
Transistor Count	25.4M CPU; 295M L3
Process	0.18u CMOS, 6 metal layer
Package	Organic Land Grid Array
Machine Width	6 insts/clock (4 ALU/MM, 2 Ld/St, 2 FP, 3 Br)
Registers	14 ported 128 GR & 128 FR; 64 Predicates
Speculation	32 entry ALAT, Exception Deferral
Branch Prediction	Multilevel 4-stage Prediction Hierarchy
FP Compute Bandwidth	3.2 GFlops (DP/EP); 6.4 GFlops (SP)
Memory -> FP Bandwidth	4 DP (8 SP) operands/clock
Virtual Memory Support	64 entry ITLB, 32/96 2-level DTLB, VHPT
L2/L1 Cache	Dual ported 96K Unified & 16KD; 16KI
L2/L1 Latency	6 / 2 clocks
L3 Cache	4MB, 4-way s.a., BW of 12.8 GB/sec;
System Bus	2.1 GB/sec; 4-way Glueless MP Scalable to large (512+ proc) systems

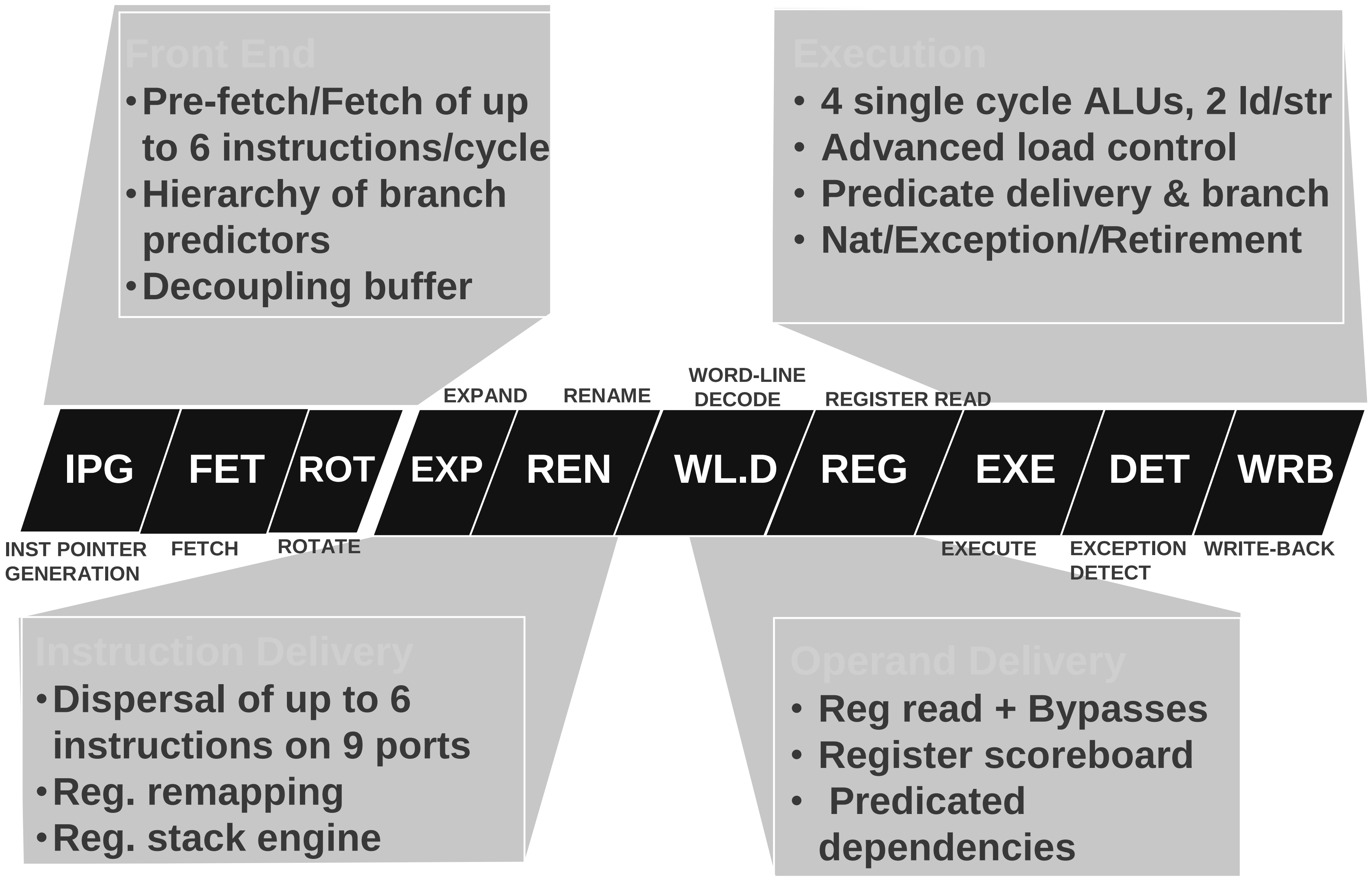
Itanium™ EPIC Design Maximizes SW-HW Synergy

(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)



Front End
- Pre-fetch/Fetch of up to 6 instructions/cycle
- Hierarchy of branch predictors
- Decoupling buffer

Execution
- 4 single cycle ALUs, 2 ld/str
- Advanced load control
- Predicate delivery & branch
- Nat/Exception//Retirement

Instruction Delivery
- Dispersal of up to 6 instructions on 9 ports
- Reg. remapping
- Reg. stack engine

Operand Delivery
- Reg read + Bypasses
- Register scoreboard
- Predicated dependencies

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

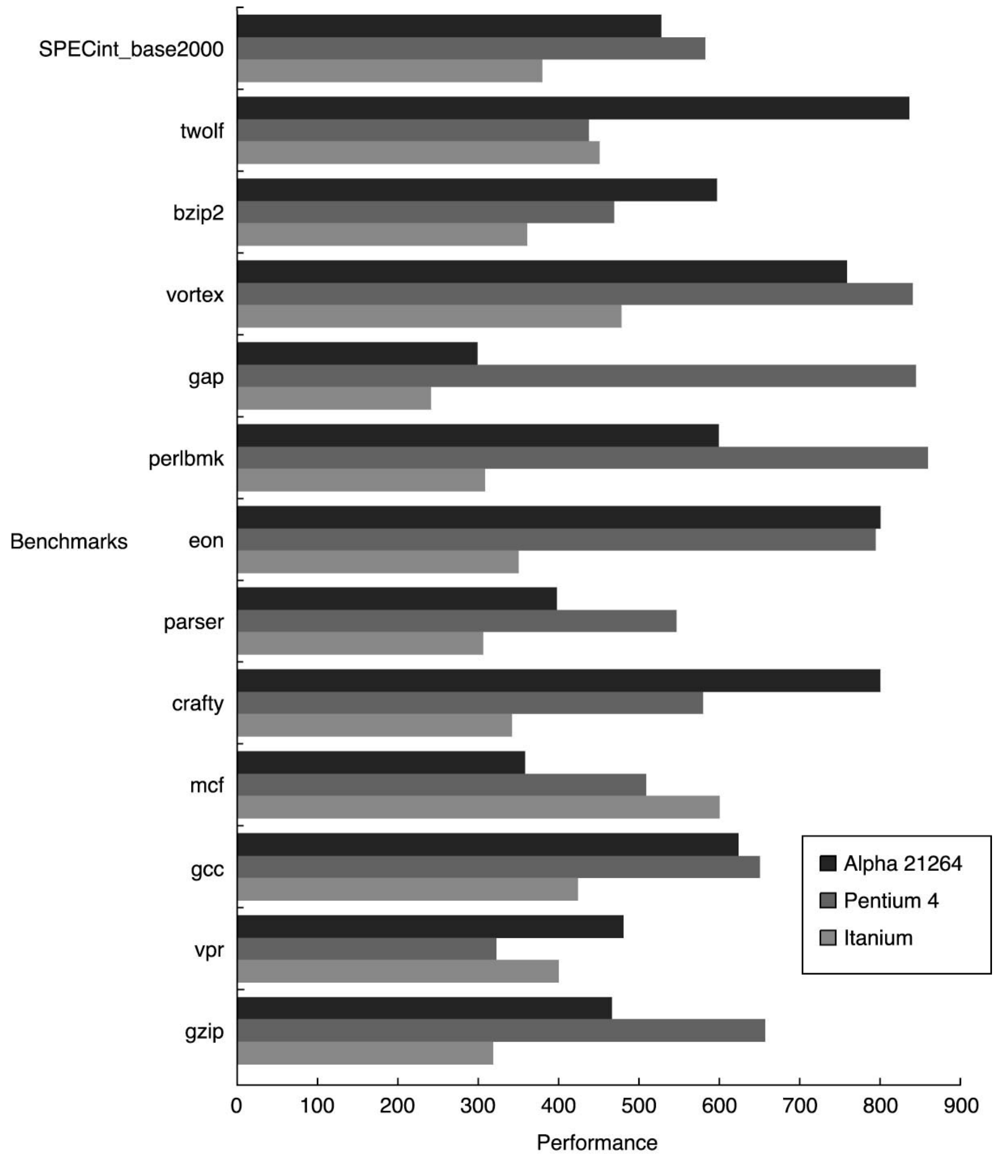
Comments on Itanium

- 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!

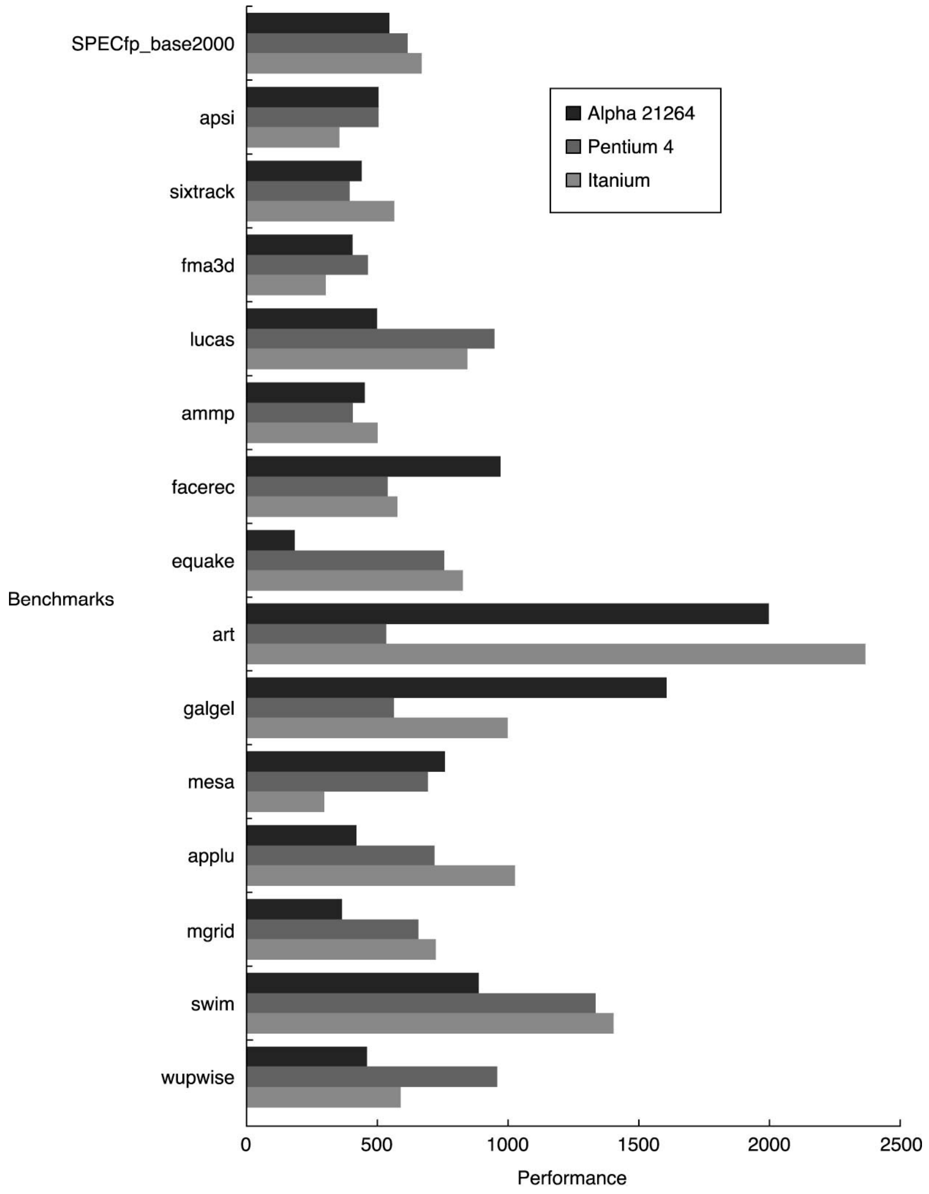
Peformance of IA-64 Itanium?

- Despite the existence of silicon, no significant standard benchmark results are available for the Itanium
- Whether this approach will result in significantly higher performance than other recent processors is unclear
- The clock rate of Itanium (800 MHz) is competitive but slower than the clock rates of several dynamically-scheduled machines, which are already available, including the Pentium III, Pentium 4 and AMD Athlon

SPECint


SPECint_base2000
twolf
bzip2
vortex
gap
perlbmk
Benchmarks
eon
parser
crafty
mcf
gcc
vpr
gzip
0
100
200
300
400
500
600
700
800
900
Performance
Alpha 21264
Pentium 4
Itanium

SPECfp



SPECfp_base2000
apsi
sixtrack
fma3d
lucas
ammp
facerec
equake
art
galgel
mesa
applu
mgrid
swim
wupwise
Benchmarks
Alpha 21264
Pentium 4
Itanium
0
500
1000
1500
2000
2500
Performance

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 Sceduling 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