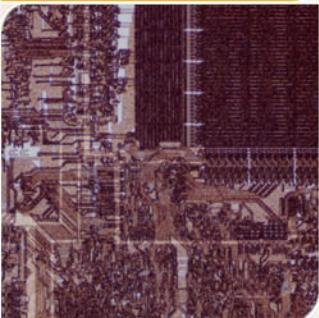


CS4290/CS6290

Fall 2011

Prof. Hyesoon Kim



**Georgia
Tech**

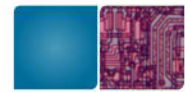


College of
Computing

Thanks to Prof. Loh & Prof. Prvulovic !



PREDICATED EXECUTION

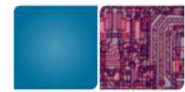


Predicated Instructions

- Instructions are predicated
 - > Depending on the predicate value the instruction is valid or becomes a No-op.

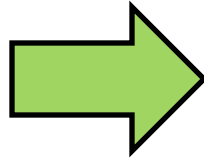
(p) add R1 = R2 + R3

P	R1 = R2 + R3
TRUE	R1 <- R2 + R3
FALSE	No op



If-conversion

```
If ( a == 0 ) {  
    b = 1;  
}  
else {  
    b = 0;  
}
```



Set p

(p) b = 1

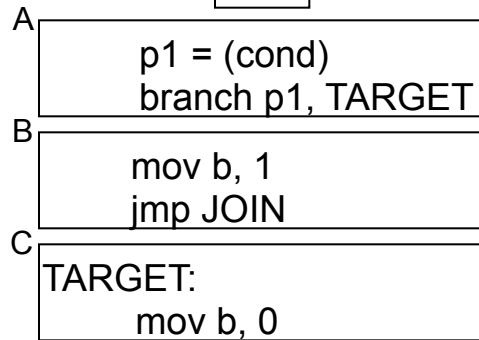
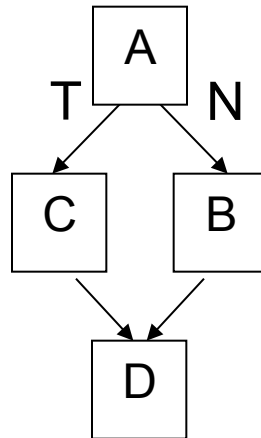
(!p) b = 0

Branch Prediction vs. Predicated Execution

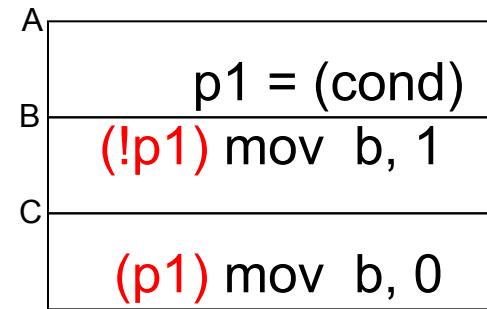
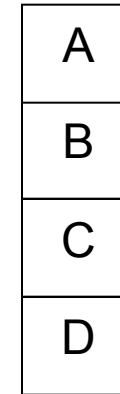
```

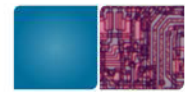
if (cond) {
    b = 0;
}
else {
    b = 1;
}
    
```

(normal branch code)



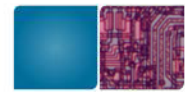
(predicated code)





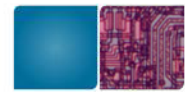
Benefit of Predicated Execution

- Eliminate branch mispredictions
 - Convert control dependency to data dependency
- Increase compiler's optimization opportunities
 - Trace scheduling, bigger basic blocks, instruction re-ordering
 - SIMD (Nvidia G80), vector processing



Limitations

- More **machine resources**
 - Fetch more instructions
 - Occupy useful resources (ROB, scheduler..)
- **ISA** should support predicated execution
 - (ISA), predicate registers
 - X86: c-move
- In OOO, supporting predicated execution is harder
 - Three input sources
 - Dependent instructions cannot be executed.

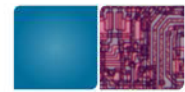


C-move

- Conditional move
 - The simplest form of predicated execution
 - Works only for registers not for memory
 - E.g.) CMOVA r16, r/m16 (move if CF=0 and ZF=0)
- Full predication support
 - Only IA-64 (later lecture)

Think think think... (Research questions...)

- When to use predicated execution?
 - Hard to predict?
 - Short branches?
 - Compiler optimization benefit?
- Who should decide it?
- Applicable to all branches?
 - Loop, function calls, indirect branches ...



STATIC INSTRUCTION SCHEDULING



Data-Dependence Stalls w/o OOO

- Multiple-Issue (Superscalar), but *in-order*
 - Instructions executing in same cycle cannot have RAW
 - Limits on WAW

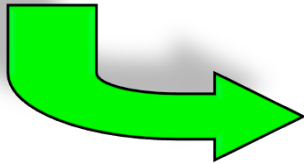


Solutions: Static Exploitation of ILP

- Code Transformations
 - Code scheduling, loop unrolling, tree height reduction, trace scheduling
- VLIW

Simple Loop Example

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



```
Loop: L.D      F0,0(R1)           ; F0 = array element
      ADD.D   F4,F0,F2           ; add scalar in F2
      S.D     F4,0(R1)          ; store result
      DADDUI  R1,R1,#-8         ; decrement pointer
                                      ; 8 bytes (per DW)
      BNE     R1, R2, Loop      ; branch R1 != R2
```

Assume:

Single-Issue	
FP ALU → Store	+2 cycles
Load DW → FP ALU	+1 cycle
Branch	+1 cycle

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

Scheduled Loop Body

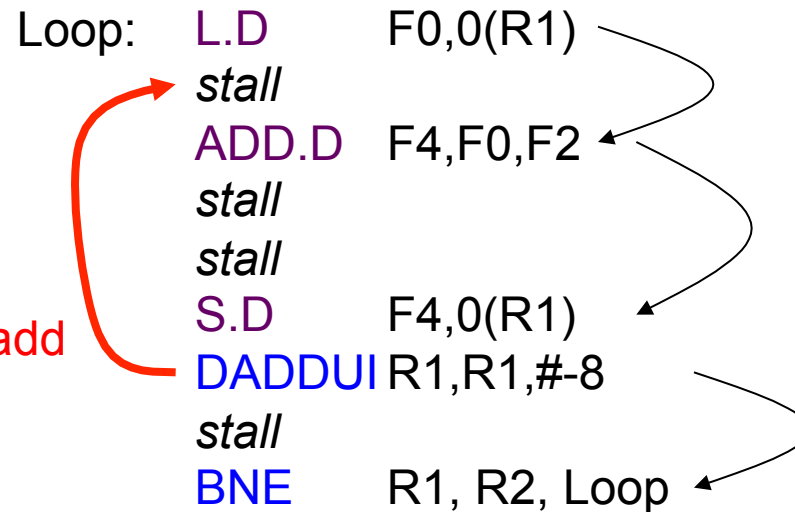
Assume:

FP ALU → Store +2 cycles

Load DW → FP ALU +1 cycle

Branch +1 cycle

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



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

Scheduling for Multiple-Issue

A: $R1 = R2 + R3$

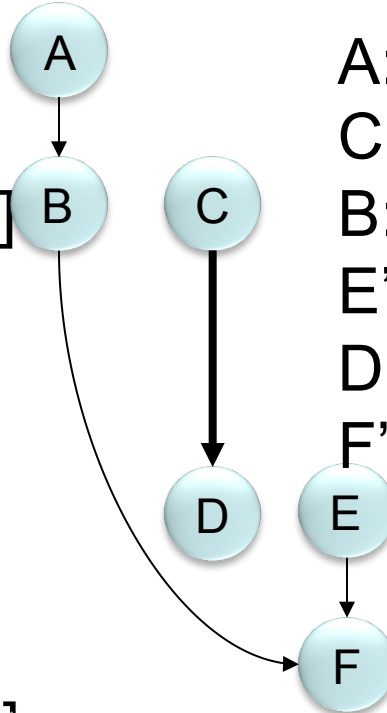
B: $R4 = R1 - R5$

C: $R1 = \text{LOAD } 0[R7]$

D: $R2 = R1 + R6$

E: $R6 = R3 + R5$

F: $R5 = R6 - R4$



A: $R1 = R2 + R3$

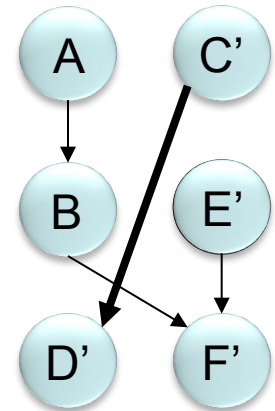
C': $R8 = \text{LOAD } 0[R7]$

B: $R4 = R1 - R5$

E': $R9 = R3 + R5$

D': $R2 = R8 + R6$

F': $R5 = R9 - R4$



A: $R1 = R2 + R3$

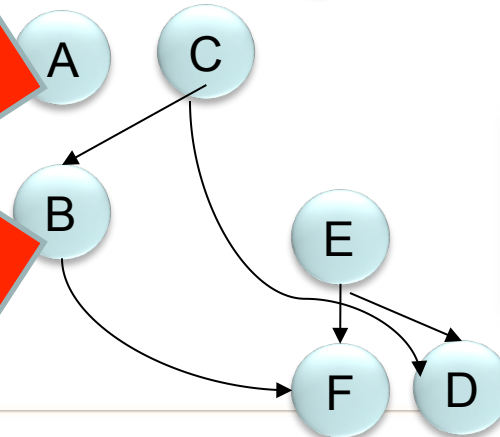
C: $R1 = \text{LOAD } 0[R7]$

B: $R4 = R1 - R5$

E: $R6 = R3 + R5$

D: $R2 = R1 + R6$

F: $R5 = R6 - R4$

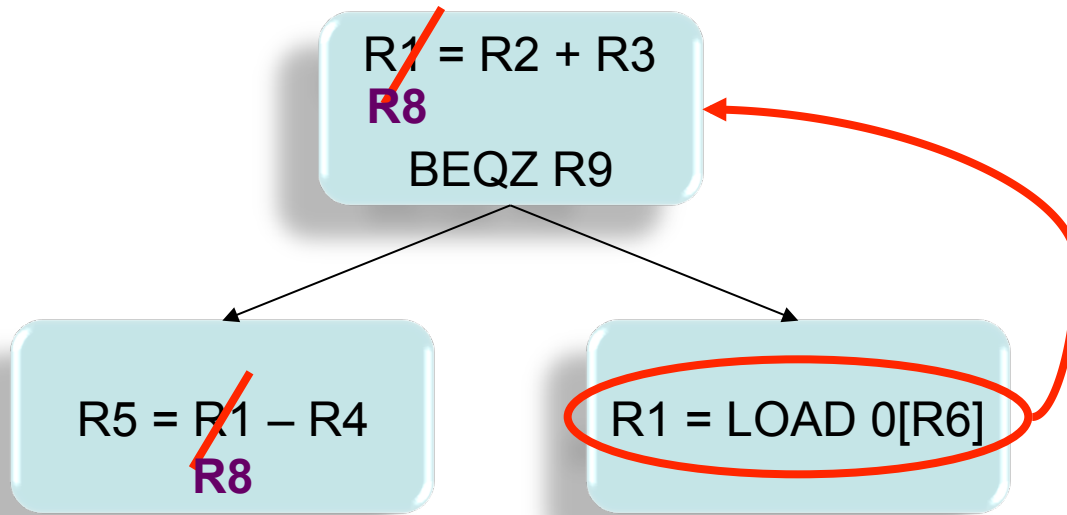


Same functionality,
no stalls



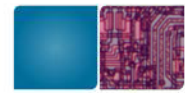
Interaction with RegAlloc and Branches

- Largely limited by architected registers
 - weird interactions with register allocation ... could possibly cause more spills/fills
- Code motion may be limited:



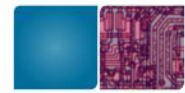
Need to allocate registers differently

Causes unnecessary execution of LOAD when branch goes left (AKA Dynamic Dead Code)



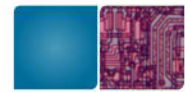
Sidetrack: Register Spills/Fills

- Register allocations: start with high-level assignment. Using Pseudo registers.
 - Pseudo registers \rightarrow ISA register (# of registers is bounded)
- Register spill/fill
 - Not enough registers: “spill” to memory
 - Need spilled contents: “fill” from memory
- Want to minimize fills and spills



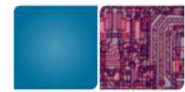
Goal of Multi-Issue Scheduling

- Place as many independent instructions in sequence
 - “as many” → up to execution bandwidth
 - Don’t need 7 independent insts on a 3-wide machine
 - Avoid pipeline stalls
- If compiler is really good, we should be able to get high performance on an in-order superscalar processor
 - In-order superscalar provides execution B/W, compiler provides dependence scheduling



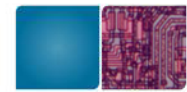
Why this Should Work

- Compiler has “all the time in the world” to analyze instructions
 - Hardware must do it in $< 1\text{ns}$
- Compiler can “see” a lot more
 - Compiler can do complex inter-procedural analysis, understand high-level behavior of code and programming language
 - Hardware can only see a small number of instructions at a time: increase hardware complexity



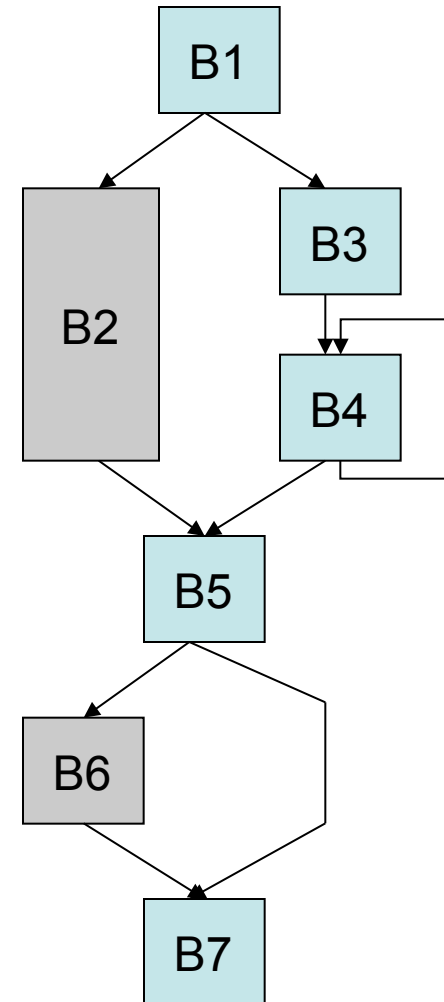
Why this Might not Work

- Compiler has limited access to dynamic information
 - Profile-based information
 - Perhaps none at all, or not representative
 - Ex. Branch T in 1st ½ of program, NT in 2nd ½, looks like 50-50 branch in profile
 - No program phase, control path
- Compiler has to generate static code
 - Cannot react to dynamic events like data cache misses

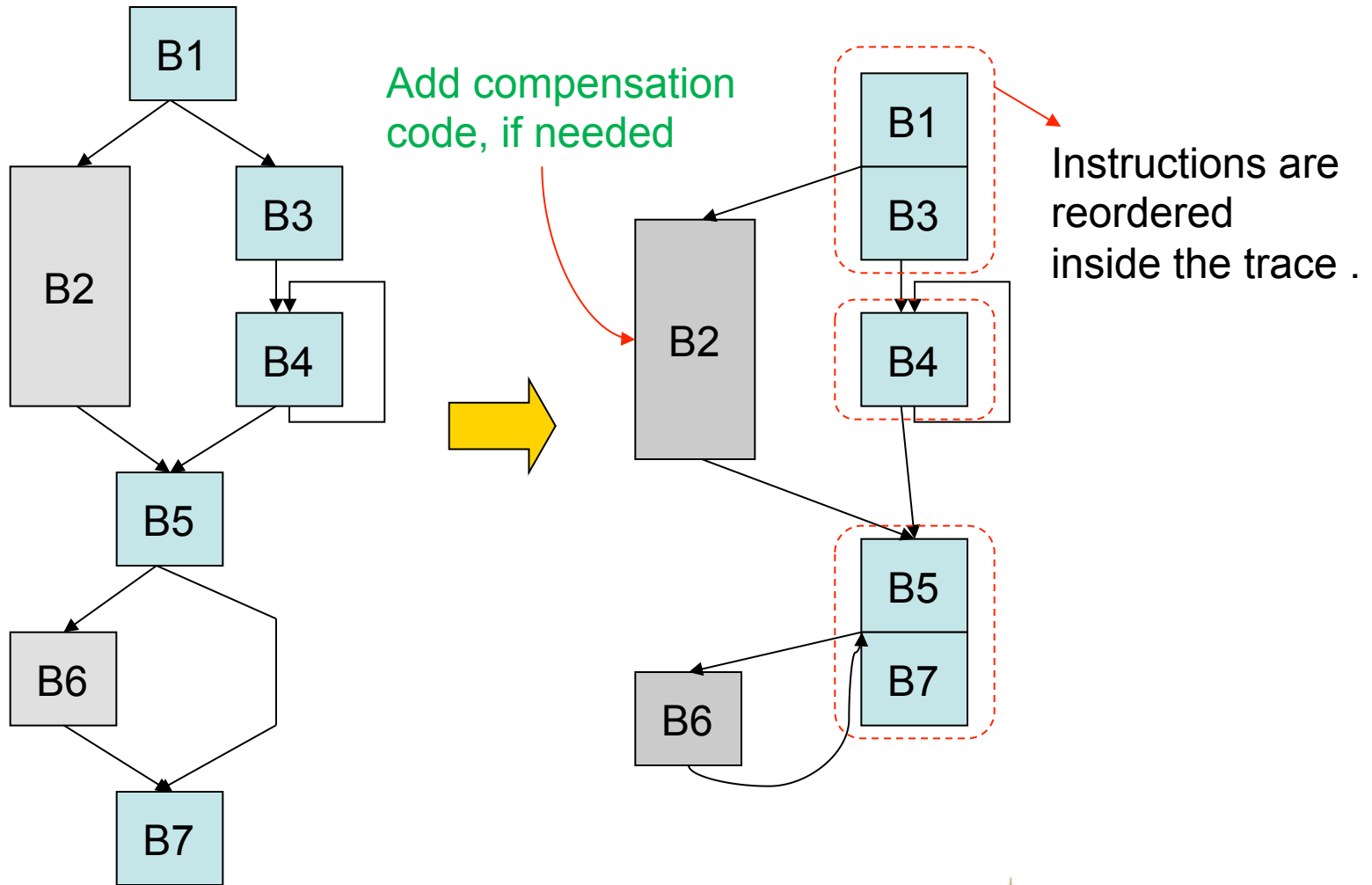


Trace

- Trace
 - Sequence of instructions
 - Including branches
 - Not including loops
- B1,B3,B4,B5,B7 is the most frequently executed path
 - Three traces in this path
 - B1,B3
 - B4
 - B5,B7



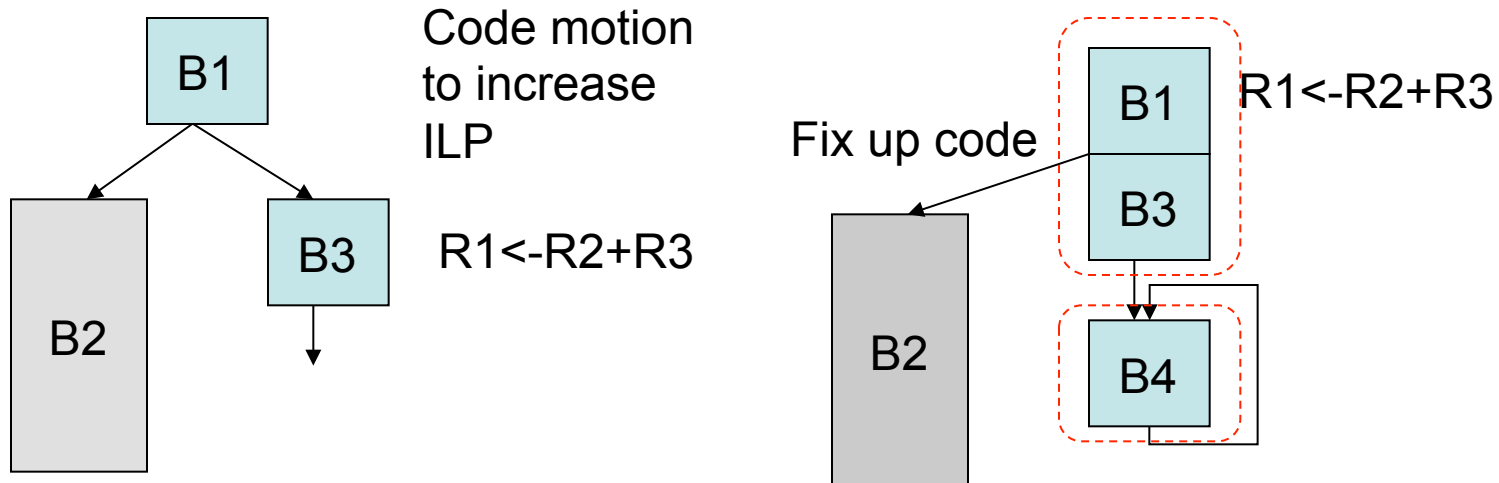
Trace Selection & Trace Computation

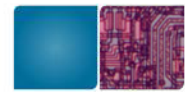


Trace Scheduling



- Basic Idea
 - Increase ILP along the important execution path by removing constraints due to the unimportant path.





Trace Scheduling

- Works on all code, not just loops
 - Take an execution trace of the common case
 - Schedule code as if it had no branches
 - Check branch condition when convenient
 - If mispredicted, clean up the mess
- How do we find the “common case”
 - Program analysis or profiling



Trace Scheduling Example

```
a=log(x) ;  
if (b>0.01) {  
    c=a/b;  
}else{  
    c=0;  
}
```



```
a=log(x) ;  
c=a/b;  
y=sin(c) ;  
if (b<=0.01)  
    goto fixit;
```

```
fixit:  
    c=0;  
    y=0; // sin(0)
```

```
y=sin(c) ;
```

Suppose profile says
that $b > 0.01$
90% of the time

Now we have larger basic block
for our scheduling and optimizations



Pay Attention to Cost of Fixing

- Assume the code for $b > 0.01$ accounts for 80% of the *time*
- Optimized trace runs 15% faster
- But, fix-up code may cause the remaining 20% of the time to be even slower!
- Assume fixup code is 30% slower

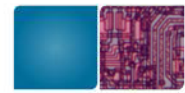
By Amdahl's Law:

$$\text{Speedup} = 1 / (0.2 + 0.8 \cdot 0.85) \\ = 1.176$$

= + 17.6% performance

$$\text{Speedup} = 1 / (0.2 \cdot 1.3 + 0.8 \cdot 0.85) \\ = 1.110$$

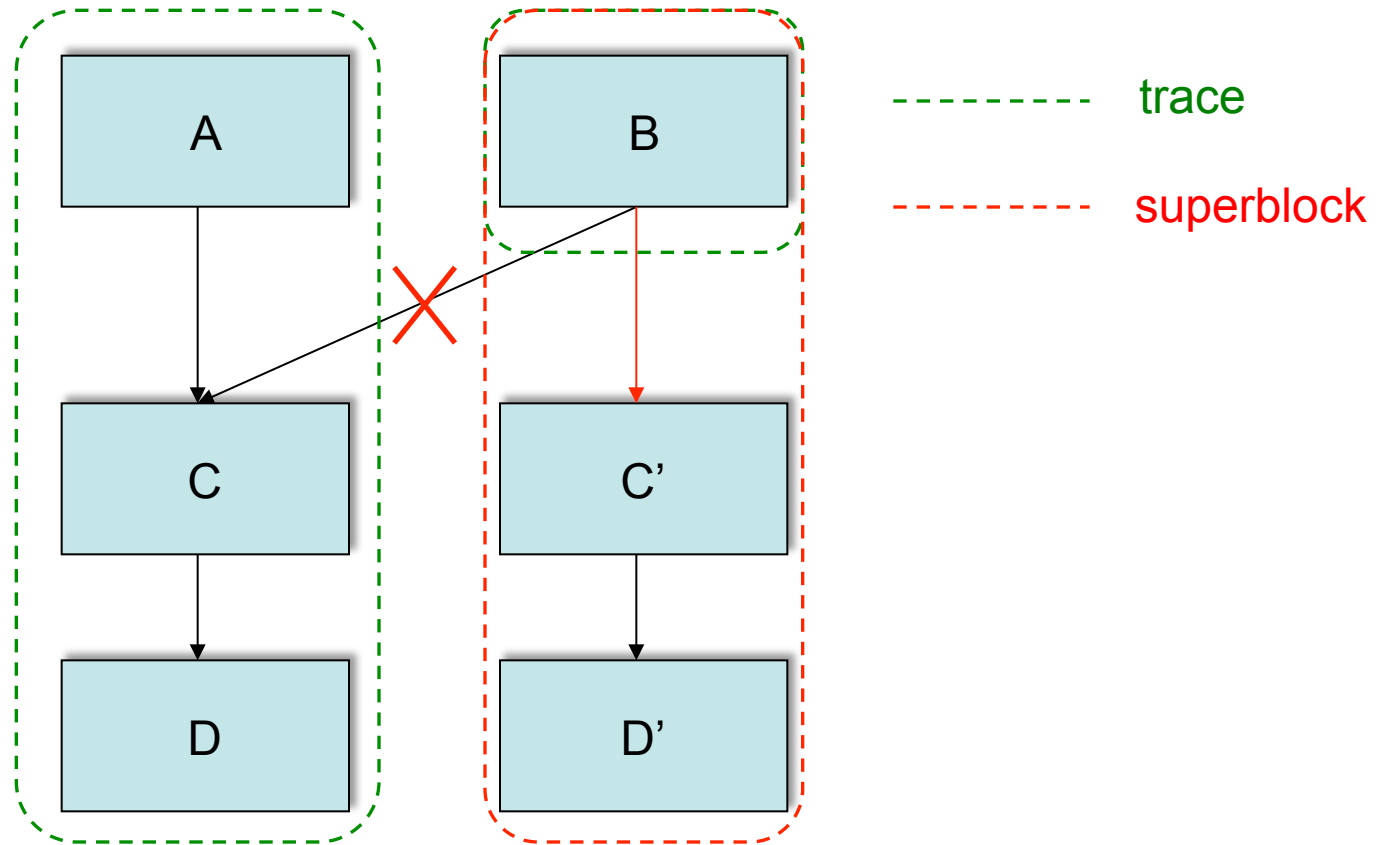
Over 1/3 of the benefit removed!

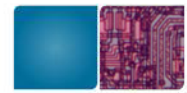


Superblocks

- Superblock removes problems associated with side entrances
- Superblock
 - A trace which has no side entrances.
 - Control may only enter from the top but may leave at one or more exit points.
 - Traces are identified using execution profile information.
 - Using tail duplication to eliminate side entrances

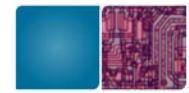
Example of tail duplication.



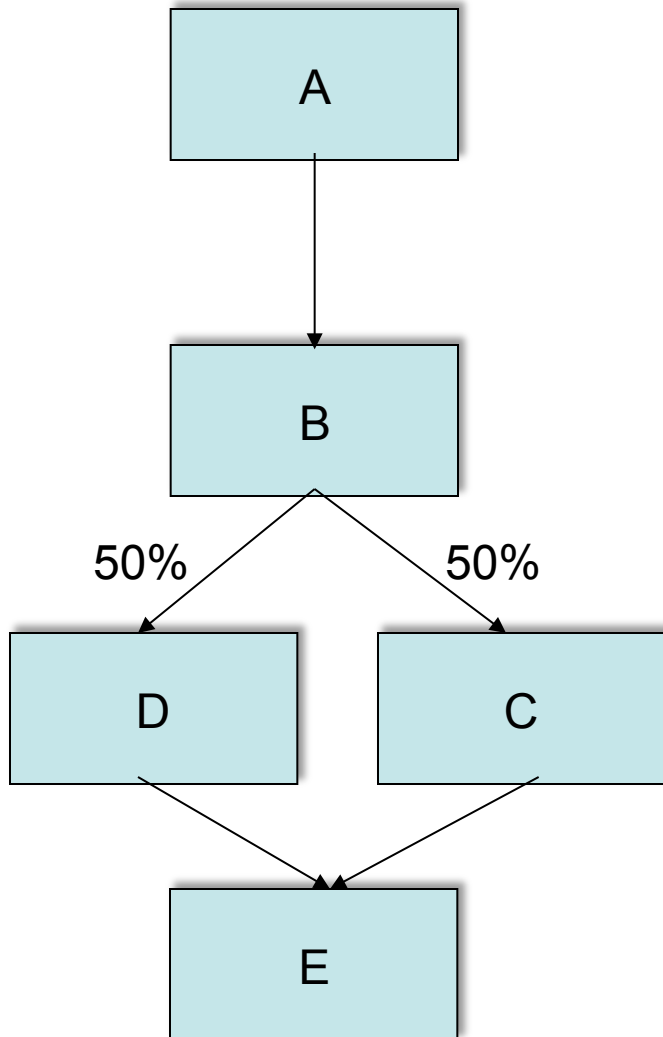


Superblock

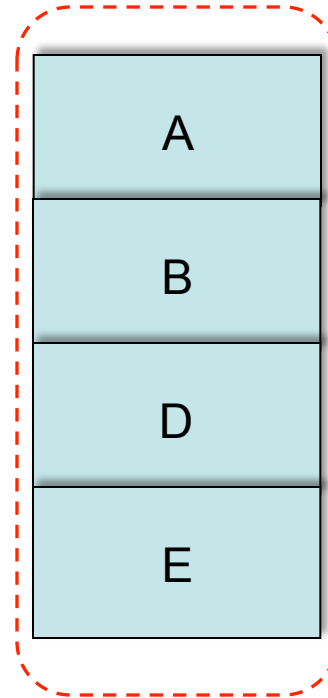
- Enlarge a block size
 - Loop unrolling
 - Loop peeling
- Global code scheduling
- Code bloat?



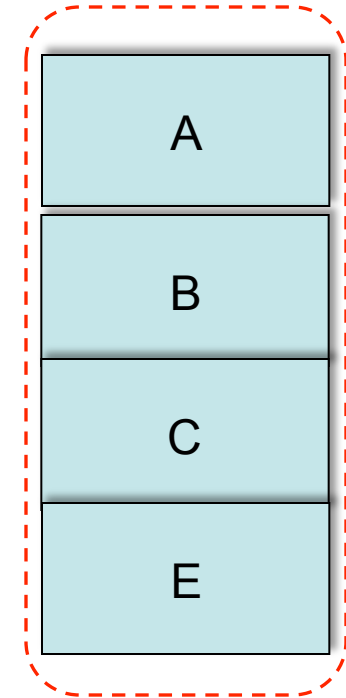
Hyperblock Formation



trace

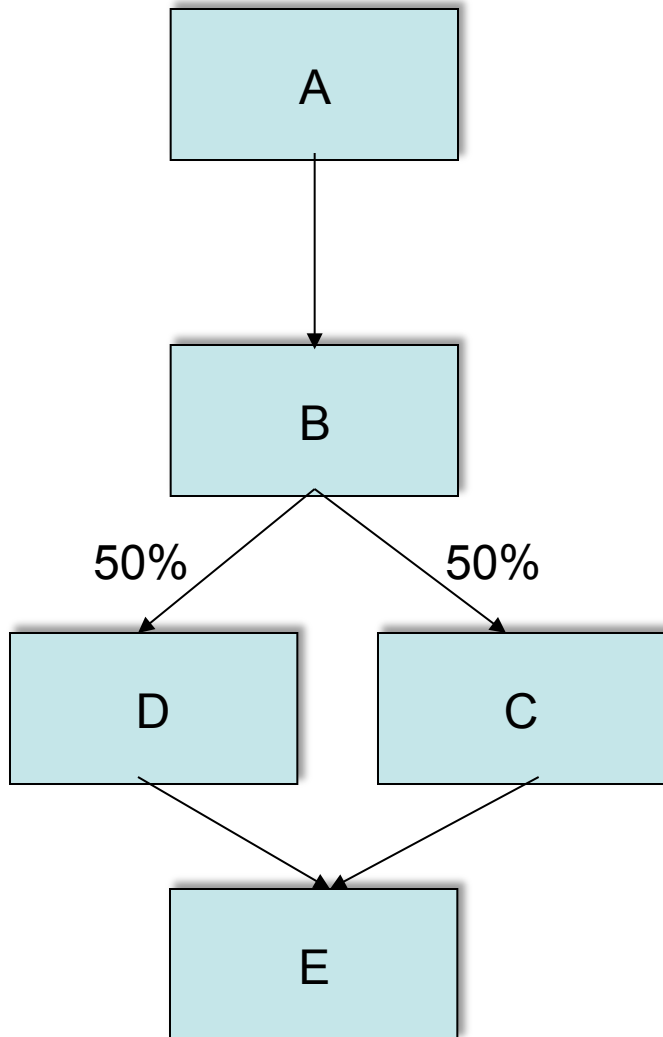


trace

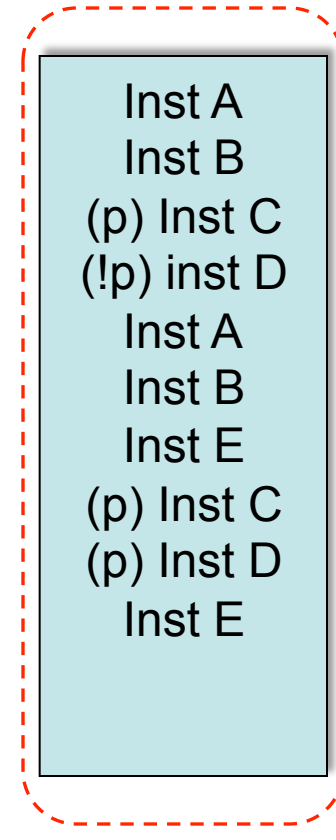
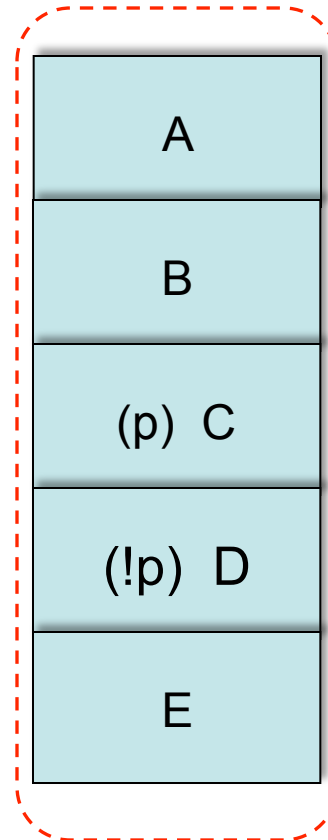


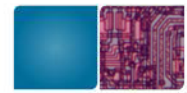


Hyperblock Formation & Scheduling



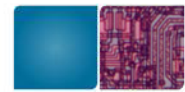
Hyperblock





Hyperblock

- Hyperblock scheduling
 - Combine basic blocks from multiple paths of control (using if-conversion)
 - For programs without heavily biased branches, hyperblocks provide a more flexible framework

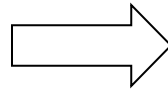


OTHER COMPILER TECHNIQUES

Loop Unrolling

- Transforms an M-iteration loop into a loop with M/N iterations
 - We say that the loop has been unrolled N times

```
for (i=0; i<100; i++)  
    a[i]*=2;
```



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

Some compilers can do this (gcc -funroll-loops)
Or you can do it manually (above)



Unrolling Often Not Enough

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

Unroll

`for (i=0; i<100; i+=2) {`
`prod*=a[i];`
`prod*=a[i+1];`
`}`

Loop: LD F0, 0(R1)
 MUL F7, F7, F0
 ADD R1, R1, 8
 BNE R1, R2, Loop

Loop: LD F0, 0(R1)
 LD F1, 8(R1)
 MUL F7, F7, F0
 MUL F7, F7, F1
 ADD R1, R1, 16
 BNE R1, R2, Loop

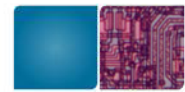
- Need a lot of unrolling to hide load latency
- Muls also slow and critical

Software Pipelining: The Idea



- Instruction pipelining:
 - Each stage performs different operation on a different instruction
 - Stage 4 writes back instruction i
 - Stage 3 does memory access for instruction $i+1$
 - Stage 2 executes instruction $i+2$
 - Stage 1 decodes instruction $i+3$
 - Stage 0 fetches instruction $i+4$
- Software pipelining:
 - Each instruction in the loop body executes operations from different logical iterations of the loop

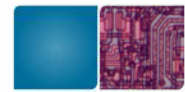
Software Pipelining



```
for (i=0; i<100; i++)  
    sum+=a[i]*b[i];
```

- We want to
 - Load a[i] and b[i], then after some time
 - Do the multiply, then after some time
 - Do the add to sum
- Software pipeline “stages”
 - Stage 1: Loads
 - Stage 2: Multiply
 - Stage 3: Add to sum

Software Pipelining



```
for (i=0; i<100; i++)  
    sum+=a[i]*b[i];
```



```
for (i=0; i<100; i++)  
{  
    a0 = a[i]; b0 = b[i];  
    prod = a0 * b0;  
    sum += prod  
}
```

Assume:
LOAD – 3 CPI
MUL – 3 CPI
ADD – 1 CPI

Software Pipelining



```
for (i=0; i<100; i++)  
    sum+=a[i]*b[i];
```



```
for (i=0; i<100; i+=2)  
{  
    a0 = a[i];    b0 = b[i];  
    a1 = a[i+1]; b1 = b[i+1];  
    prod0 = a0 * b0;  
    prod1 = a1 * b1;  
    sum += prod0  
    sum += prod1  
}
```

Assume:
LOAD – 3 CPI
MUL – 3 CPI
ADD – 1 CPI

Software Pipelining



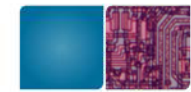
```
for (i=0; i<100; i++)  
    sum+=a[i]*b[i];
```



```
for (i=0; i<100; i+=2)  
{  
    a0 = a[i];    b0 = b[i];  
    a1 = a[i+1]; b1 = b[i+1];  
    prod0 = a0 * b0;  
    prod1 = a1 * b1;  
    sum += prod0  
    sum += prod1  
}
```

Assume:
LOAD – 3 CPI
MUL – 3 CPI
ADD – 1 CPI

Software Pipelining



```
for (i=0 ; i<100 ; i++)  
    sum+=a [i] *b [i] ;
```



Assume:
LOAD – 3 CPI
MUL – 3 CPI
ADD – 1 CPI

```
    p2=a [0] *b [0] ;  
    a1=a [1] ;b1=b [1] ;  
    for (i=2 ; i<100 ; i++) {  
        sum+=p2 ;  
        p2=a1*b1 ;  
        a1=a [i] ;b1=b [i] ;  
    }  
    sum+=p2 ;  
    sum+=a1*b1 ;
```

prolog

Start-up: Stages 1-2 for iter 0
 Stage 1 for iter 1

Pipeline: Stage 3 for iter i-2,
 Stage 2 for iter i-1,
 Stage 1 for
 iter i

kernel

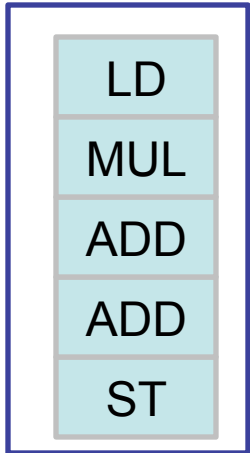
Finish-up: Stage 2 for it 98,
 Stages 2 and 3 for iter 99

epilog

Why?

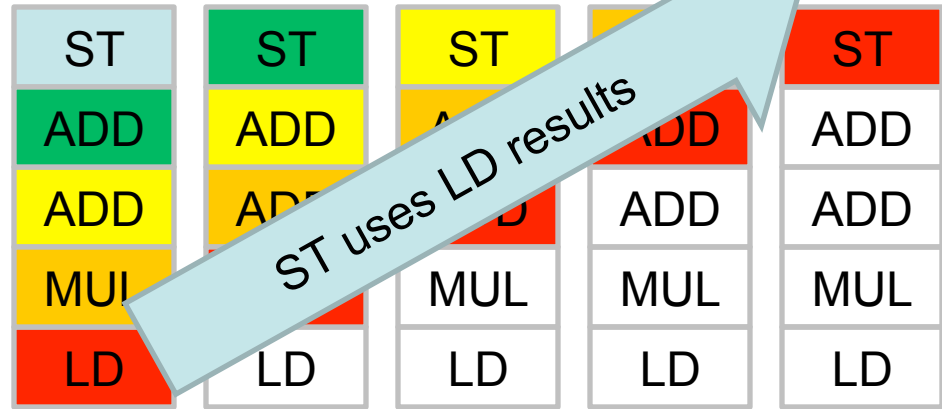
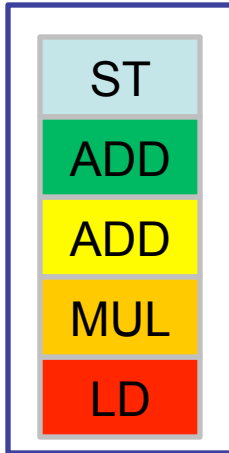


Original code



pipelined code

iter i-4
iter i-3
iter i-2
iter i-1
iter i



Cycle

N

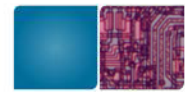
N+1

N+2

N+3

N+4

Question



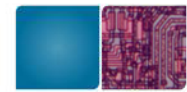
```
LOOP LD F0, 0 (R1)
      ADD F4, F0, F2
      SD F4, 0 (R1)
      DADDUI R1, R1, #-8
      BNE R1, R2, LOOP
```

Assume:
LOAD – 3 CPI
MUL – 3 CPI
ADD – 1 CPI

Show a software-pipelined version of this loop. Assume that you have infinite number of registers. Include start-up and clean-up code.

Answer





Function Inlining

- Sort of like “unrolling” a function
- Similar benefits to loop unrolling:
 - Remove function call overhead
 - CALL/RETN (and possible branch mispreds)
 - Argument/ret-val passing, stack allocation, and associated spills/fills of caller/callee-save regs
 - Larger block of instructions for scheduling
 - If-conversion is possible
- Similar problems
 - Increase register pressure
 - Primarily code bloat

```
main()
{
  ....
  c=max(a,b);
  ....
  c=max(a,b);
}
max(a,b)
{
  If (a>=b) return a;
  else return b;
}
```



```
main()
{
  ....
  If (a>=b) c=a;
  else c = b
  ....
  If (a>=b) c=a;
  else c = b;
  ...
}
```

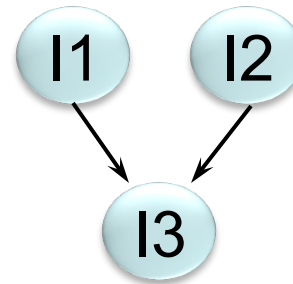


Tree Height Reduction

- Shorten critical path(s) using associativity

ADD R6 , R2 , R3
 ADD R7 , R6 , R4
 ADD R8 , R7 , R5

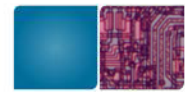
ADD R6 , R2 , R3
 ADD R7 , R4 , R5
 ADD R8 , R7 , R6



Not all Math operations are associative!

$$R8 = ((R2 + R3) + R4) + R5$$

$$R8 = (R2 + R3) + (R4 + R5)$$

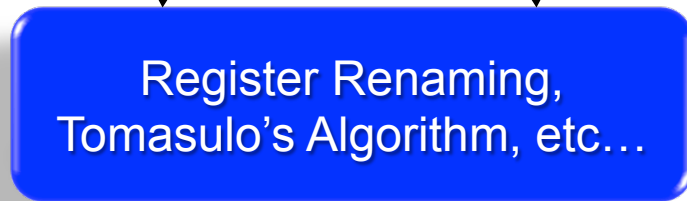


VLIW

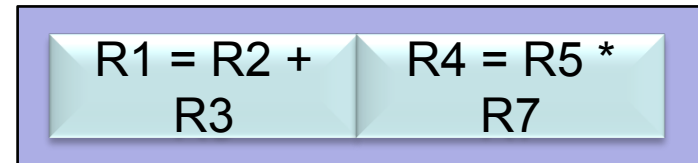


Parallelism/Dependencies Explicit

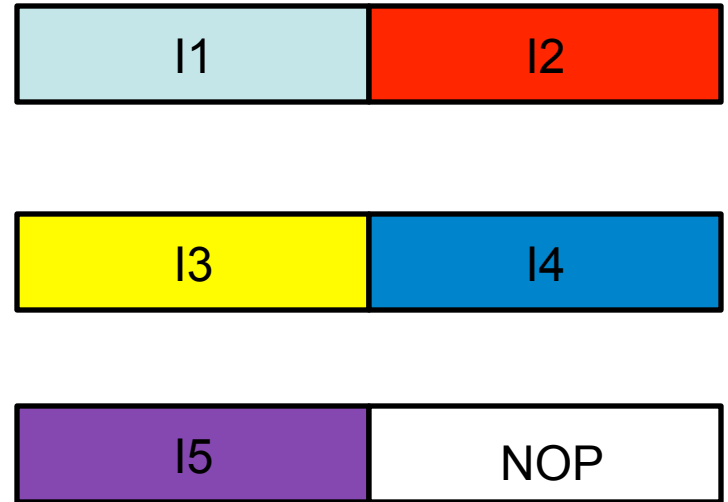
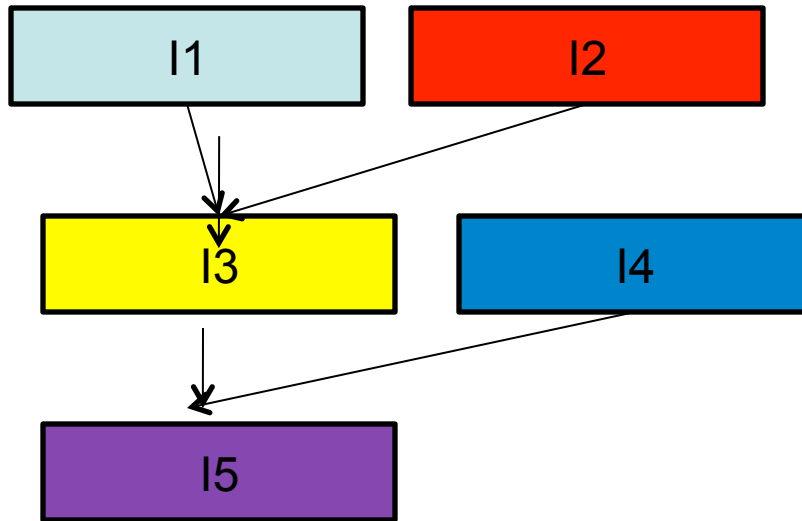
- Compiler can do analysis to find independent instructions
 - Rather than having Tomasulo-like hardware to detect such instructions
- Directly communicate this to the HW



Yup, they're independent



Static Instruction Scheduling



VLIW



- VLIW = Very Long Instruction Word

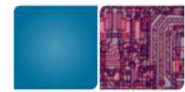


- **Everything** is statically scheduled
 - All hardware resources exposed to compiler
 - Compiler must figure out what to do and when to do it
 - Get rid of complex scheduling hardware
 - More room for “useful” resources
- Examples:
 - Texas Instruments DSP processors
 - Transmeta’s processors
 - Intel IA-64 (EPIC)



Why is VLIW good?

- Let the compiler do all of the hard work
 - Expose functional units, bypasses, latencies, etc.
 - Compiler can do its best to schedule code well
 - Compiler has plenty of time to do analysis
 - Compiler has larger scope (view of the program)
- Works extremely well on regular codes
 - Media Processing, Scientific, DSP, etc.
- Can be energy-efficient
 - Dynamic scheduling hardware is power-hungry



Why is VLIW hard?

- Latencies are not constant
 - Statically scheduled assuming fixed latencies
- Irregular applications
 - Dynamic data structures (pointers)
 - “Common Case” changes when input changes
- Code can be very large
 - *Every resource exposed* also means that instructions are “verbose”, with fields to tell each HW resource what to do
 - Many, many “NOP” fields
- 3wide VLIW machine → 6 wide VLIW machine?
- Where is instruction parallelism?

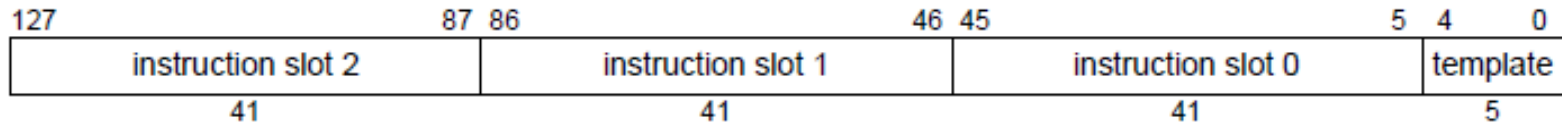


Extreme Example: Intel IA-64 (EPIC)

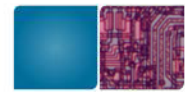
- Goal: Keep the best of VLIW, fix problems
 - Keep HW simple and let the compiler do its job
 - Support to deal with non-constant latencies
 - Make instructions more compact
- The reality
 - Compiler still very good at regular codes
 - HW among the most complex ever built by Intel
 - Good news: compiler still improving



IA-64 Bundles

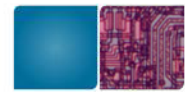


- Bundle == The “VLIW” Instruction
 - 5-bit template encoding
 - also encodes “stops”
 - Three 41-bit instructions
- 128 bits per bundle
 - average of 5.33 bytes per instruction
 - x86 only needs 3 bytes on average



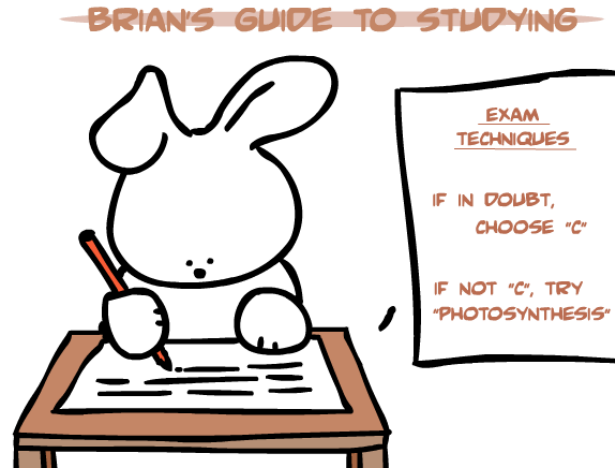
IA-64 Groups

- Compiler assembles *groups* of instructions
 - No register data dependencies between insts in the same group
 - Memory deps may exist
 - Compiler explicitly inserts “stops” to mark the end of a group
 - Group can be arbitrarily long



Question

- A: $R1 = R2 + R3$
- B: $R4 = R1 - R5$
- C: $R1 = \text{LOAD } 0[R7]$
- D: $R2 = R1 + R6$
- E: $R6 = R3 + R5$
- F: $R5 = R6 - R4$

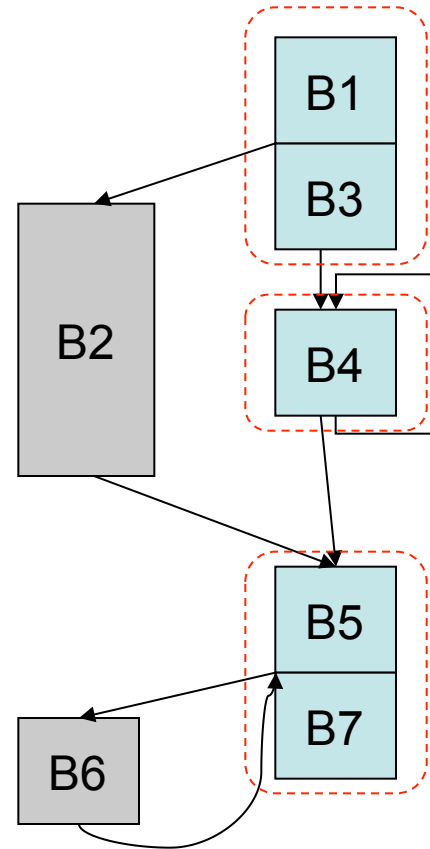
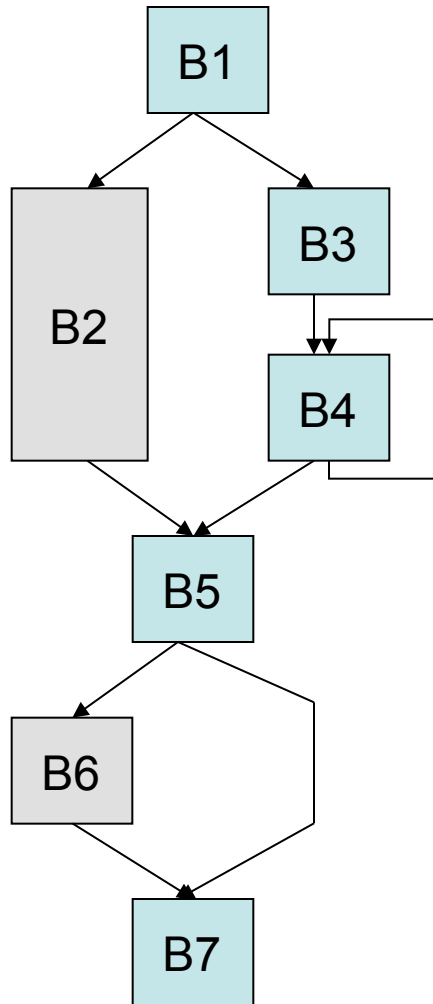


- Write 3-wide VLIW code
 - (1) All instruction has 1 CPI
 - (2) LD instruction has 2 CPI





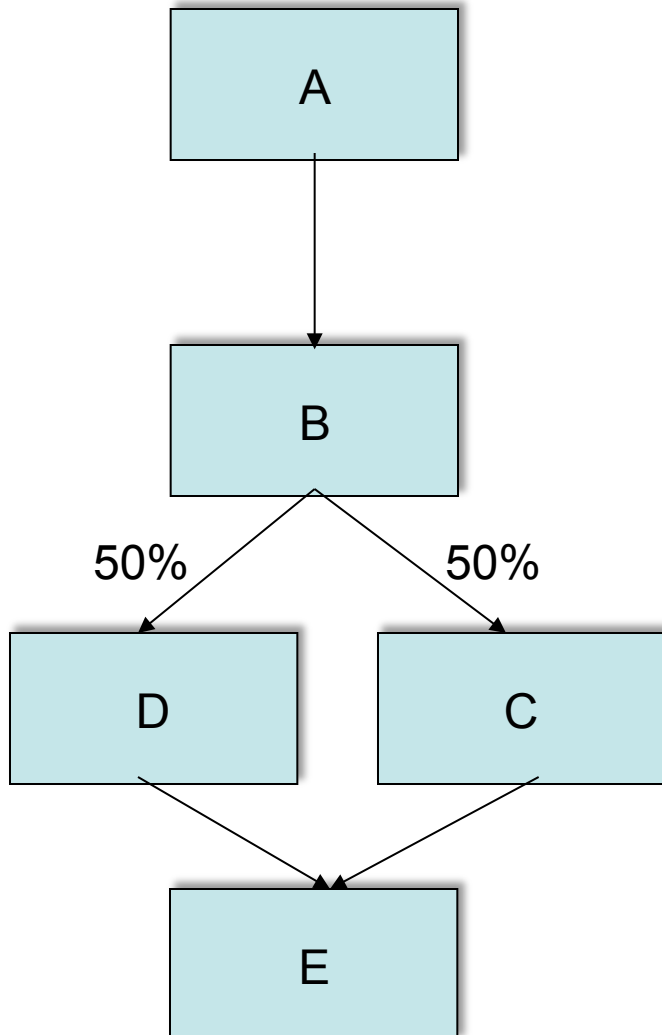
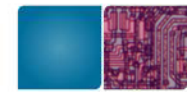
Trace Scheduling



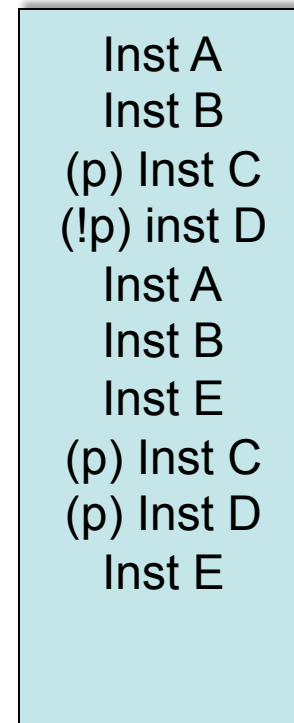
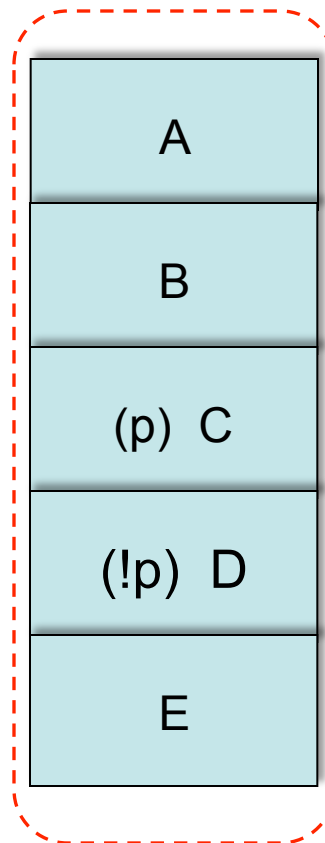
Inst 1	Inst 2
Inst 3	NOP
Inst 4	NOP

Inst 5	NOP
Inst 6	Inst 7

Hyperblock



Hyperblock

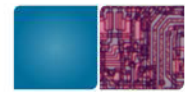


Inst 1	Inst 2
(p) Inst 3	(!p) inst 4
Inst 5	NOP
Inst 6	Inst 7

Compiler Exceptions and Speculation Support

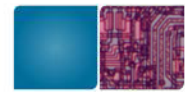


- Advanced load may trigger **exceptions** that may never happen in original code
- Solution: **speculative load does not raise exception**, it just poisons its destination reg
- The check is where the original load was
 - Check triggers a re-load if reg poisoned
 - If the exception is really supposed to happen, the (non-speculative) re-load will raise it



Data Speculation

- Why: want to schedule loads early
 - Compiler puts load early
 - Hardware starts the load early
 - Loaded value arrives in time to be used
- Problem: Exceptions ? Memory disambiguation problem ?



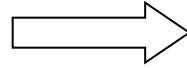
Data Speculation

- New instructions (e.g. IA-64)
 - Speculative (Advance) load and Load check
 - Hardware support for memory disambiguation problem.
- New HW
 - Advance Load Addr Table (ALAT)
or Memory Conflict Buffer (MCB)
- How it works
 - Speculative load puts data addr and dest reg into ALAT
 - Store looks for its data addr in ALAT
and *poisons* the dest regs found in matching entries
 - Check OK if register not poisoned
(if it is, recovery code loads data again)



Data Speculation Example

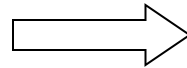
```
ST F2,100(R3)  
LD F1,0(R1)  
ADD F2,F1,F3
```



```
LD.A F1,0(R1)  
ST F2,100(R3)  
CHK.A F1  
ADD F2,F1,F3
```

- Can also do control speculation

```
BEQ R1,R2,Error  
LD F1,0(R1)  
ADD F2,F1,F3
```



```
LD.A F1,0(R1)  
BEQ R1,R2,Error  
CHK.A F1  
ADD F2,F1,F3
```


SIMD vs. VLIW

