18-640 Foundations of Computer Architecture

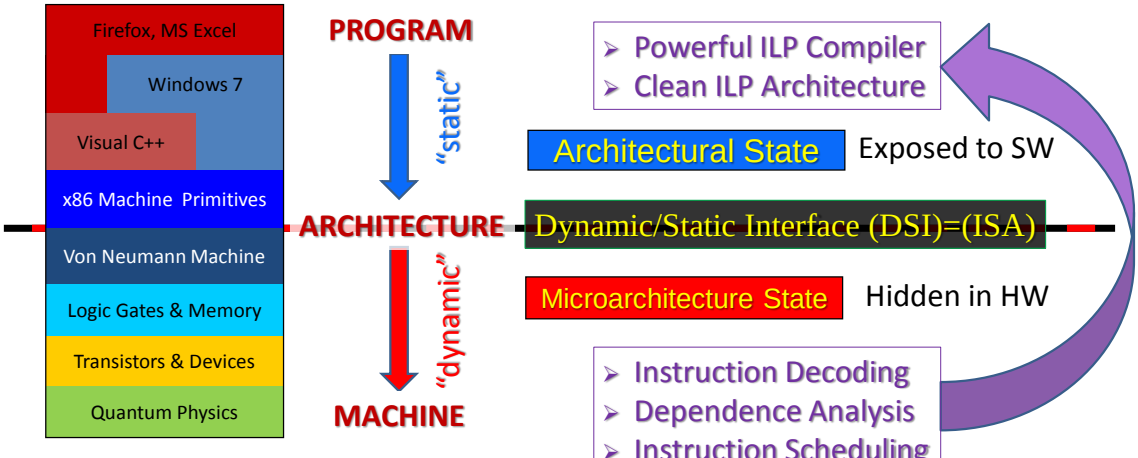
Lecture 10: "VLIW and EPIC Architectures"

John Paul Shen
September 30, 2014

- Required Reading Assignments:
  - "A VLIW Architecture for a Trace Scheduling Compiler" by Robert Colwell, et al. (1987)
  - "The Multiflow Trace Scheduling Compiler" by Geoffrey Lowney, et al. (1992)
- Recommended Reference:
  - H&P 5th Appendix H – "Hardware and Software for VLIW and EPIC" http://booksite.mkp.com/9780123838728/references.php

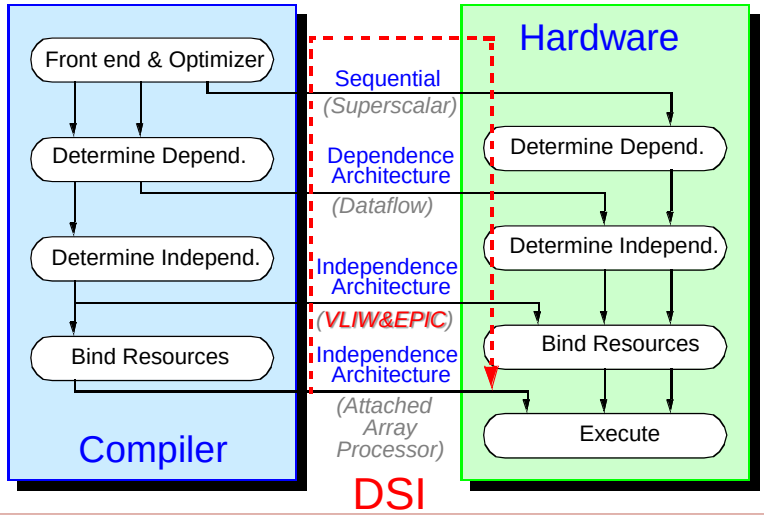
Electrical & Computer ENGINEERING

Computer Architecture: Dynamic-Static Interface



[B. Rau & J. Fisher, 1993]

HW vs. SW and Dynamic vs. Static Design Space



18-640 Foundations of Computer Architecture

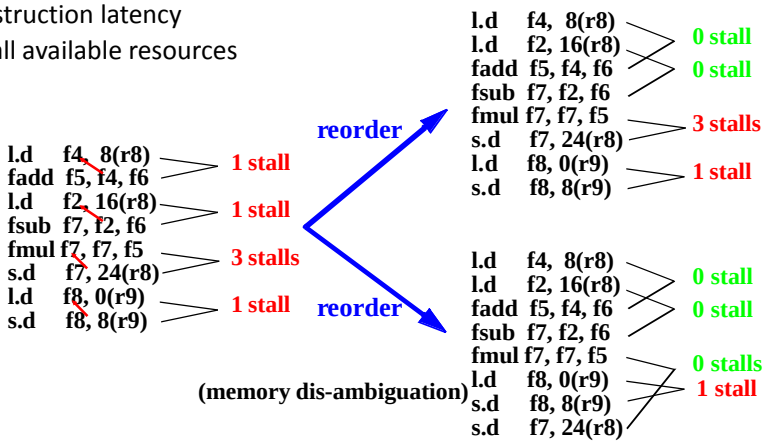
Lecture 10: "VLIW and EPIC Architectures"

A. Code Scheduling
B. Trace Scheduling
C. VLIW Architecture
D. Multiflow TRACE Computer
E. Intel IA-64 EPIC Architecture

Electrical & Computer ENGINEERING

A. Code Scheduling

- Rearrange code sequence to minimize execution time
  - Hide instruction latency
  - Utilize all available resources



Code Scheduling

- Objectives: minimize execution latency of the program
  - Start as early as possible instructions on the critical path
  - Help expose more instruction-level parallelism to the hardware
  - Help avoid resource conflicts that increase execution time
- Constraints
  - Program Precedences (Dependences)
  - Machine Resources
- Motivations
  - Dynamic/Static Interface (DSI): By employing more software (static) optimization techniques at compile time, hardware complexity can be significantly reduced
  - Performance Boost: Even with the same complex hardware, software scheduling can provide additional performance enhancement over that of unscheduled code

Precedence Constraints

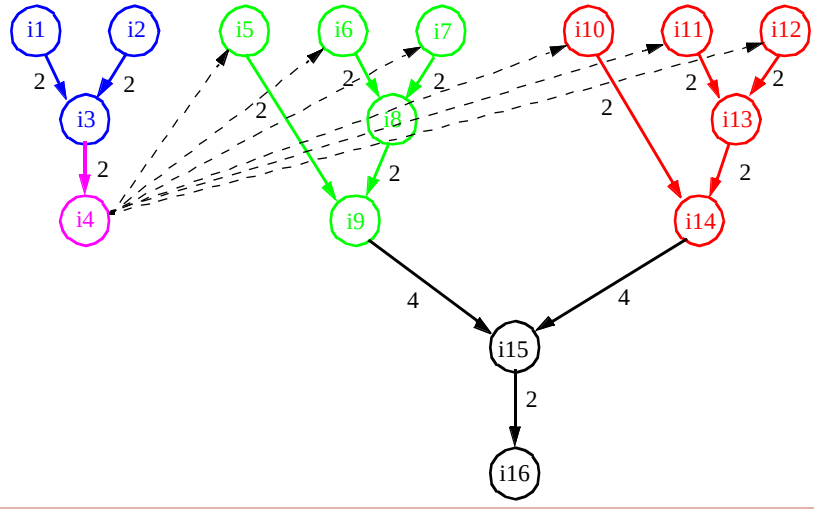
- Minimum required ordering and latency between definition and use
- Precedence Graph
  - Nodes: instructions
  - Edges (a→b): a precedes b
  - Edges are annotated with minimum latency

```
w[i+k].ip = z[i].rp + z[m+i].rp;
w[i+j].rp = e[k+1].rp*
            (z[i].rp -z[m+i].rp) -
            e[k+1].ip *
            (z[i].ip - z[m+i].ip);
```

FFT code fragment

```
i1: l.s f2, 4(r2)
i2: l.s f0, 4(r5)
i3: fadd.s f0, f2, f0
i4: s.s f0, 4(r6)
i5: l.s f14, 8(r7)
i6: l.s f6, 0(r2)
i7: l.s f5, 0(r3)
i8: fsub.s f5, f6, f5
i9: fmul.s f4, f14, f5
i10: l.s f15, 12(r7)
i11: l.s f7, 4(r2)
i12: l.s f8, 4(r3)
i13: fsub.s f8, f7, f8
i14: fmul.s f8, f15, f8
i15: fsub.s f8, f4, f8
i16: s.s f8, 0(r8)
```


Precedence Graph

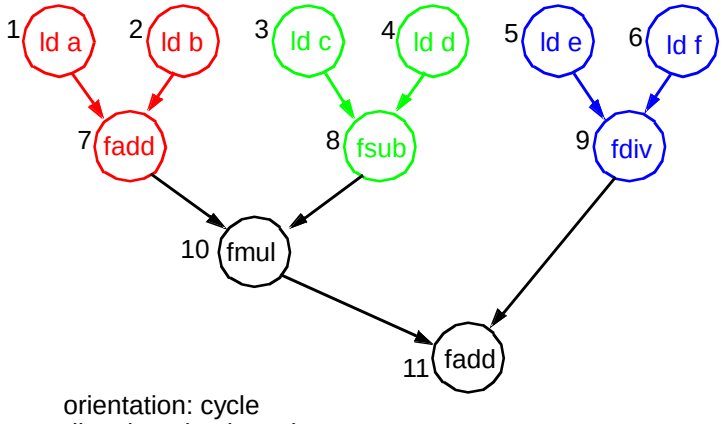


List Scheduling for Basic Blocks

- Initialize ready list that holds all ready instructions
  Ready = data ready and can be scheduled
- Choose one ready instruction *I* from ready list with the highest priority
  - Number of descendants in precedence graph
  - Maximum latency from root node of precedence graph
  - Length of operation latency
  - Ranking of paths based on importance
  - Combination of above
- Insert *I* into schedule
  Making sure resource constraints are satisfied
- Add those instructions whose precedence constraints are now satisfied into the ready list
- Can be applied in the forward or backward direction

List Scheduling Example

(a + b) * (c - d) + e/f



load: 2 cycles
add: 1 cycle
sub: 1 cycle
mul: 4 cycles
div: 10 cycles

orientation: cycle
direction: backward
heuristic: maximum latency to root

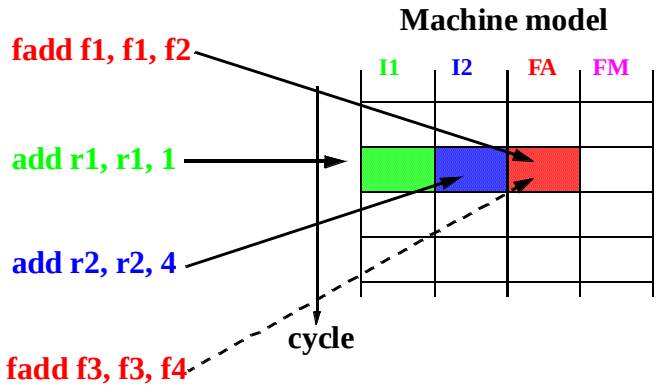
List Scheduling Example

Cycle	Ready list	Schedule	Code
1	6	6	ld f
2	5 6	5	ld e
3	4 5 6	4	ld d
4	4 9	9	fdiv (e/f)
5	3 4 9	3	ld c
6	2 3 4 9	2	ld b
7	1 2 3 4 9	1	ld a
8	1 2 8 9	8	fsub (c – d)
9	7 8 9	7	fadd (a + b)
10	9 10	10	fmul
11	9 10		nop
12	9 10		nop
13	9 10		nop
14	11	11	fadd

green *means candidate and ready*
red means candidate but not yet ready

Resource Constraints

- Bookkeeping
  - Prevent resources from being oversubscribed



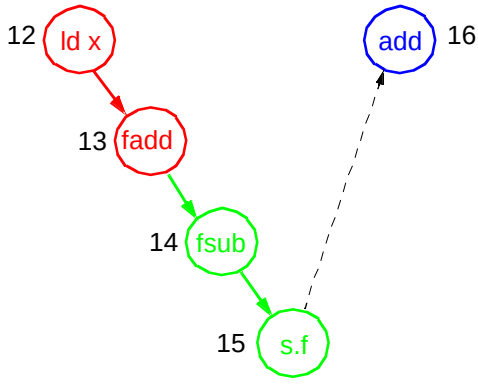
ILP List Scheduling Example

Cycle	Ready list	Schedule	Resources			Code	
			I	F	FD		
1	6	6	X			ld f	
2	5 6	5	X			ld e	
3	5 6						
4	4 9	4 9	X		X	fdiv (e/f)	ld d
5	3 4 9	3	X			ld c	
6	2 3 4 9	2	X			ld b	
7	1 2 3 4 9	1	X			ld a	
8	1 2 8 9	8		X		fsub (c – d)	
9	7 8 9	7		X		fadd (a + b)	
10	9 10	10		X		fmul	
11	9 10					nop	
12	9 10					nop	
13	9 10					nop	
14	11	11		x		fadd	

Take Home Example

- Append the following to the previous example:

```
*(p) = (x + Ry) - Rz ;
p = p + 4 ;
```



Take Home Example

Cycle	Ready list	Schedule	Resources			Code	
			I	F	FD		
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
11							
12							
13							
14							

Take Home Example

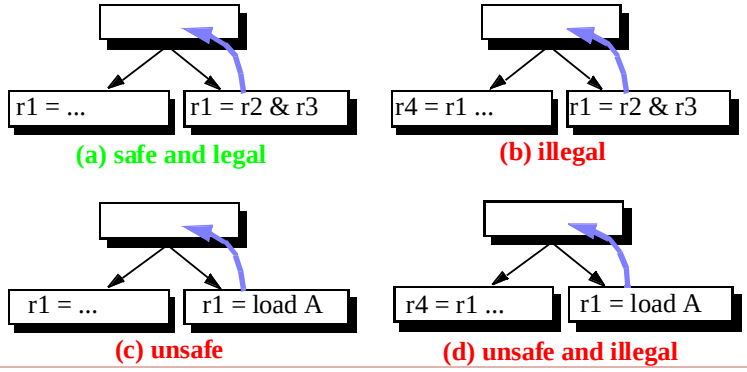
Cycle	Ready list	Schedule	Resources			Code	
			I	F	FD		
1	6	6	X			ld f	
2	5 6	5	X			ld e	
3	5 6						
4	4 9	4 9	X		X	fdiv (e/f)	ld d
5	3 4 9	3	X			ld c	
6	2 3 4 9	2	X			ld b	
7	1 2 3 4 9	1	X			ld a	
8	1 2 8 9	8		X		fsub (c – d)	
9	7 8 9 12	7 12	X	X		fadd (a + b)	ld x
10	9 10 12	10		X		fmul	
11	9 10 13	13		X		fadd	
12	9 10 14	14		X		fsub	
13	9 10 15	15	X			s.f	
14	11 16	11 16	X	X		fadd	add

Limitations of List Scheduling

- Cannot move instructions past conditional branch instructions in the program (scheduling limited by basic block boundaries)
- Problem: Many programs have small numbers of instructions (4-5) in each basic block. Hence, not much code motion is possible
- Solution: Allow code motion across basic block boundaries.
- Speculative Code Motion: "jumping the gun"
  - Execute instructions before we know whether or not we need to
  - Utilize otherwise idle resources to perform work which we speculate will need to be done
- Relies on program profiling to make intelligent decisions about speculation

Types of Speculative Code Motion

- Two characteristics of speculative code motion:
  - safety, which indicates whether or not spurious exceptions may occur
  - legality, which indicates correctness of results
- Four possible types of code motion:



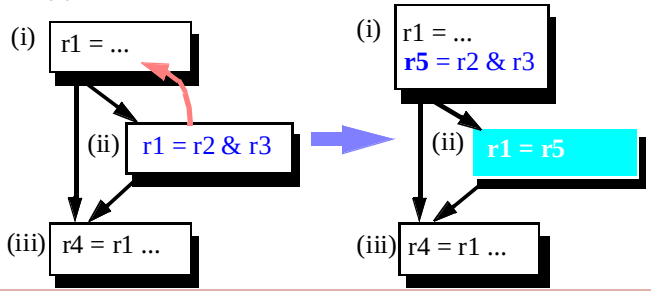
Register Renaming

- Prevents boosted instructions from overwriting register state needed on alternate execution path.
- Utilizes idle (non-live) registers (r6 in example below).

BB#	Original Code			Scheduled Code		
n		load	r4= ...		load	r4= ...
		load	r5= ...		load	r5= ...
		cmpi	c0,r4,10		cmpi	c0,r4,10
		add	r4=r4+r5		add	r4=r4+r5
		<stall>			sub	r3=r7-r4
		<stall>			and	r6=r3&r5
		bc	c0, A1		bc	c0, A1
n+1		st	... =r4		st	... =r4
n+2	A1:	sub	r3=r7-r4	A1:	st	... =r6
		and	r4=r3&r5			
		st	... =r4			

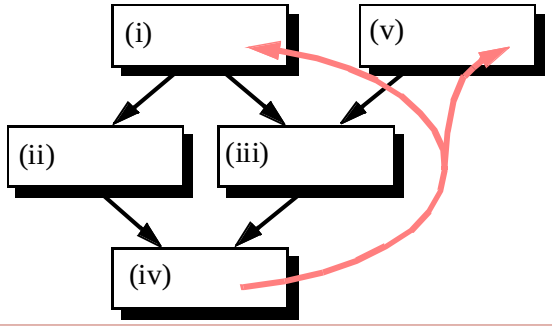
Copy Creation

- Register renaming causes a problem when there are multiple definitions of a register reaching a single use:
  - Below, definitions of r1 in both (i) and (ii) reach the use in (iii).
  - If the instruction in (ii) is boosted into (i), it must be renamed to preserve the first value of r1.
  - However, the boosted definition of r1 must reach the use in (iii) as well.
  - Hence, we insert a copy instruction in (ii).



Instruction Replication

- General case of upward code motion: crossing control flow joins.
- Instructions must be present on each control flow path to their original basic block
- Replicate set is computed for each basic block that is a source for instructions to be boosted



Profile Driven Optimizations

- Wrong optimization choices can be costly!

  How do you determine dynamic information during compilation?
- During initial compilation, "extra code" can be added to a program to generate profiling statistics when the program is executed
- Execution Profile, e.g.
  - how many times is a basic block executed
  - how often is a branch taken vs. not taken
- Recompile the program using the profile to guide optimization choices
- A profile is associated with a particular program input

  ⇒ *may not work well on all executions*

B. Trace Scheduling [Josh Fisher, 1981]

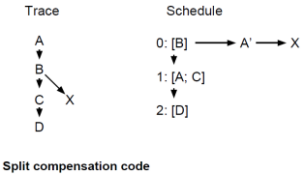
- Generate multi-basic block traces based on profiling information
  - find the most often executed control flow path
- List schedule a trace at a time
  - optimize the execution of the trace *(common case)*
  - fix any problem with off-trace paths as necessary *(infrequently executed)*
- Good for very biased and predictable branching behavior
- Trace Scheduling engendered the **VLIW architecture** innovation and was implemented in the **Multiflow TRACE compiler**, which provided the basis for ILP compilation techniques being used by Intel, HP, and DEC

Trace Scheduling Overview

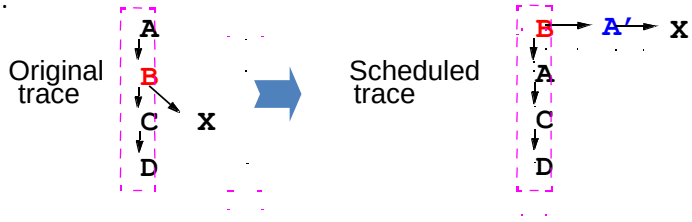
- Trace Selection
  - Select seed (the highest frequency basic block)
  - Extend trace (along the highest frequency edges)
    forward (successor of the last block of the trace)
    backward (predecessor of the first block of the trace)
  - Don't cross loop back edge
  - Bound max_trace_length heuristically
- Trace Scheduling
  - Build data precedence graph for a whole trace
  - Perform list scheduling and allocate registers
  - Add compensation code to maintain semantic correctness
- Speculative Code Motion (upward)
  - Move an instruction above branches if safe

Compensation Code for Code Motion

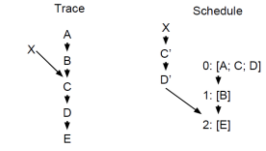
- Split Compensation Code:
  - Instruction with more than one successor



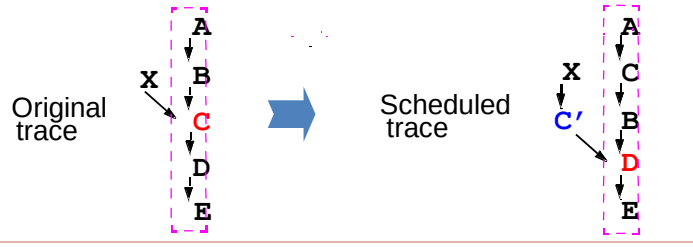
Split compensation code



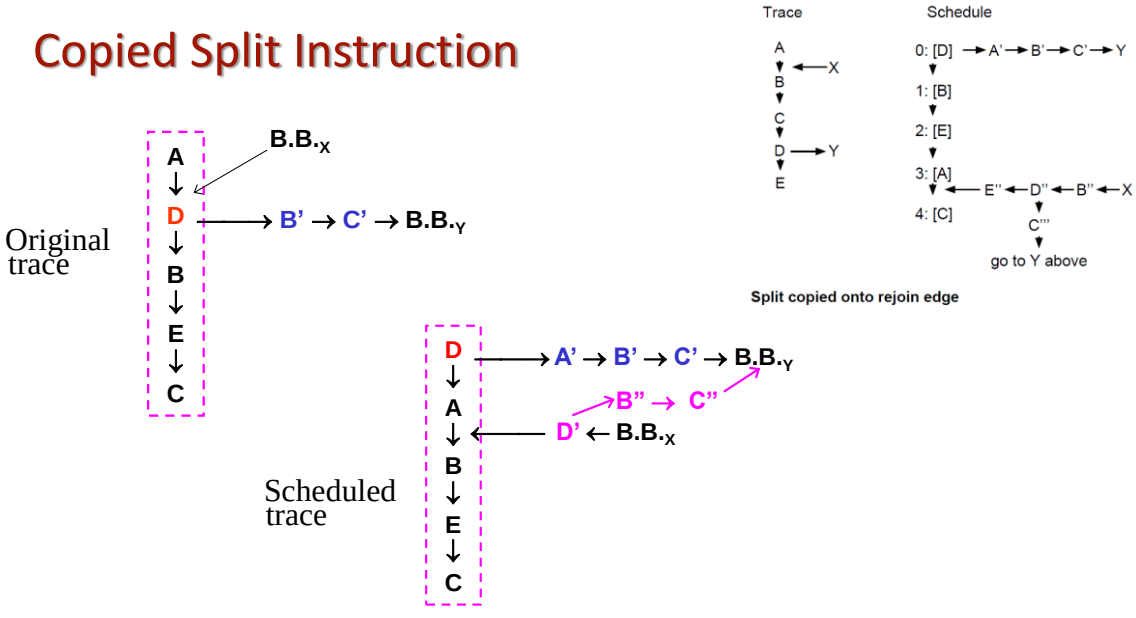
- Join Compensation Code:
  - Instruction with more than one predecessor



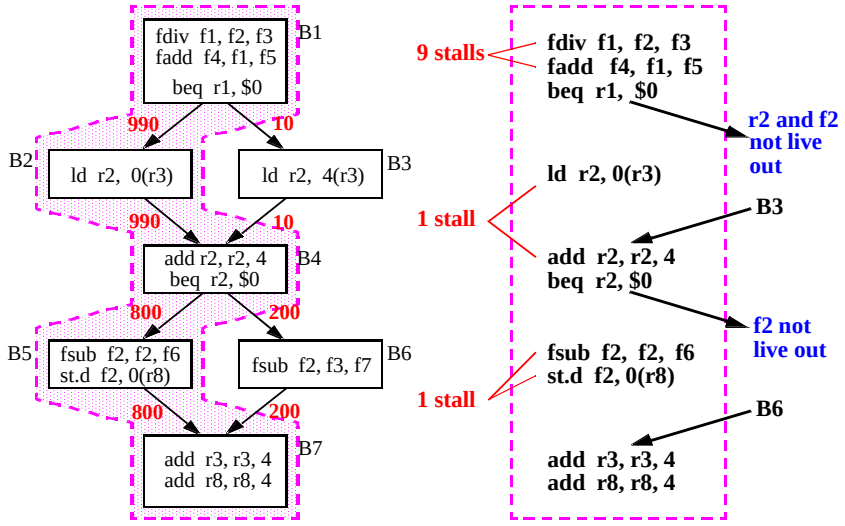
Join compensation code



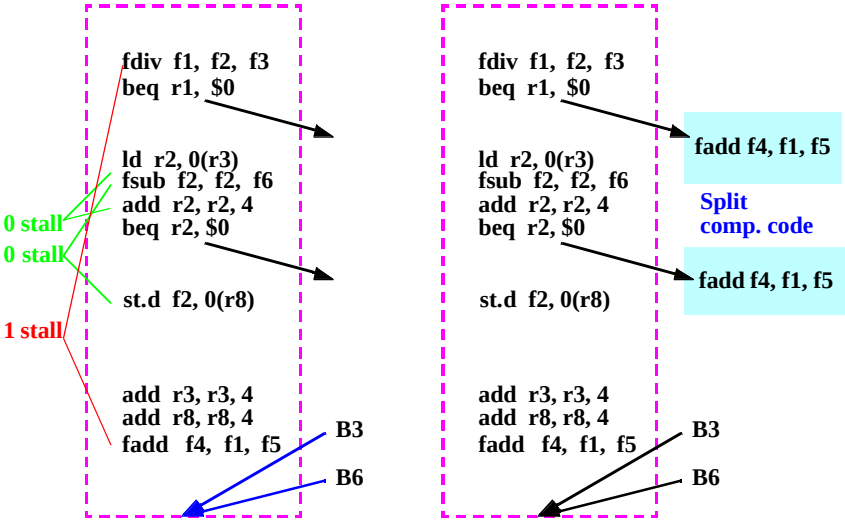
Copied Split Instruction



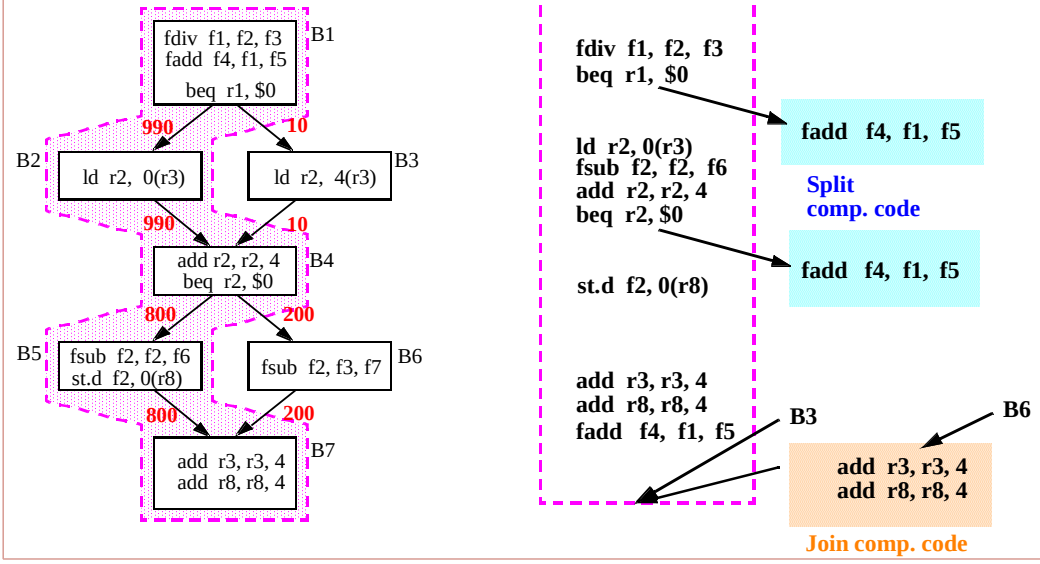
Trace Scheduling Example



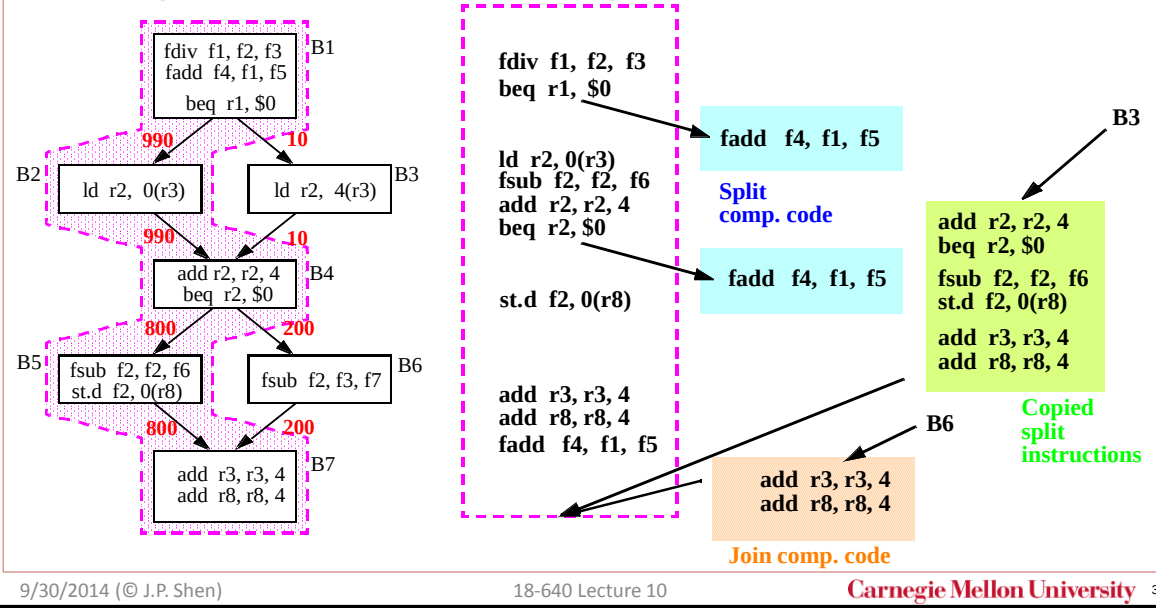
Compensation Code Example



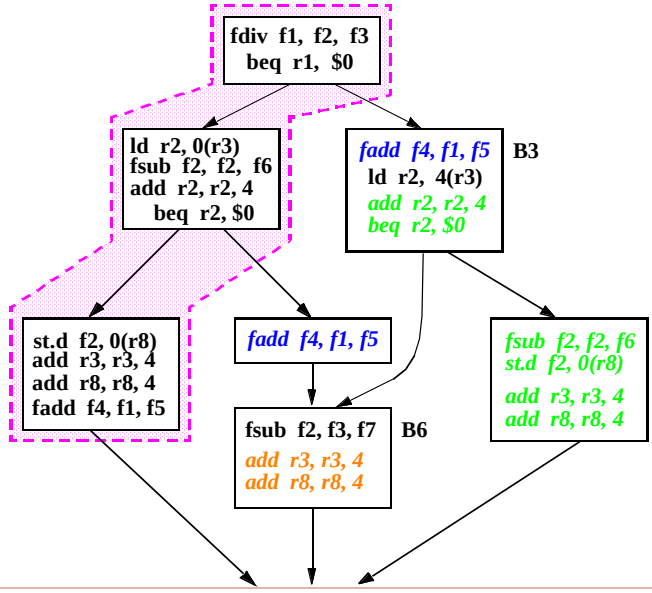
Compensation Code Example



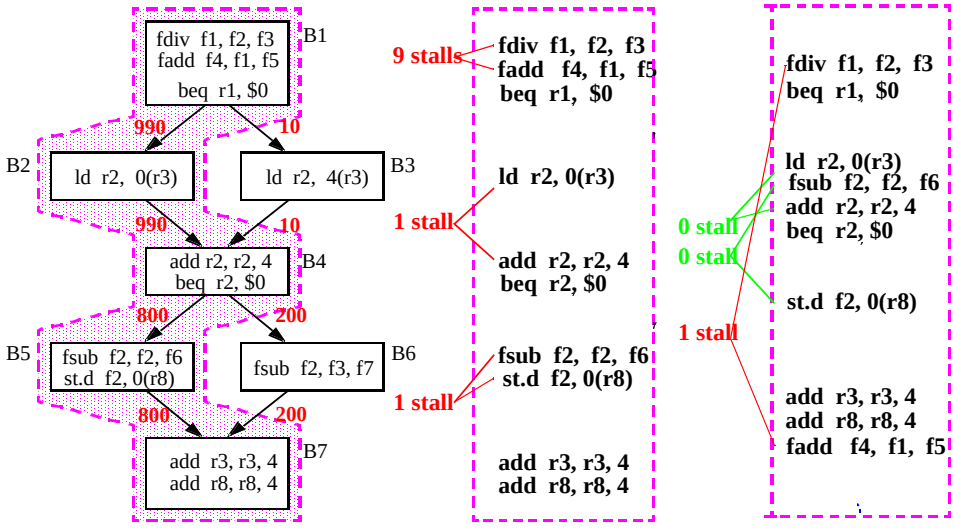
Compensation Code Example



Compensation Code Illustration

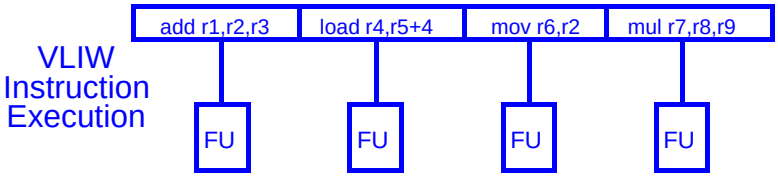


Trace Scheduling Performance Improvement



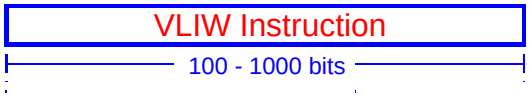
C. Very Long Instruction Word Architecture

- VLIW hardware is simple and straightforward, like SIMD machines.
- VLIW separately directs each functional unit



Principles of VLIW Operation

- Statically scheduled ILP architecture.
- Wide instructions specify many independent simple (RISC like) operations.

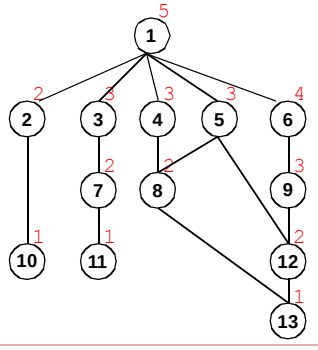


- Multiple functional units executes all of the operations in an instruction concurrently, providing fine-grain parallelism within each VLIW instruction
- Instructions directly control the hardware with no interpretation and minimal decoding
- A powerful optimizing compiler is responsible for locating and extracting ILP from the program and for scheduling operations to exploit the available parallel resources

The processor does not make any run-time control decisions below the program level

VLIW List Scheduling

- Assign Priorities
- Compute Data Ready List - all operations whose predecessors have been scheduled.
- Select from DRL in priority order while checking resource constraints
- Add newly ready operations to DRL and repeat for next instruction



4-wide VLIW				Data Ready List
1				{1}
6	3	4	5	{2,3,4,5,6}
9	2	7	8	{2,7,8,9}
12	10	11		{10,11,12}
13				{13}

Strengths of VLIW Technology

- Parallelism can be exploited at the instruction level
  - Available in both vectorizable and sequential programs
- Hardware is regular and straightforward
  - Most hardware is in the datapath performing useful computations
  - Instruction issue costs scale approximately linearly

Potentially very high clock rate

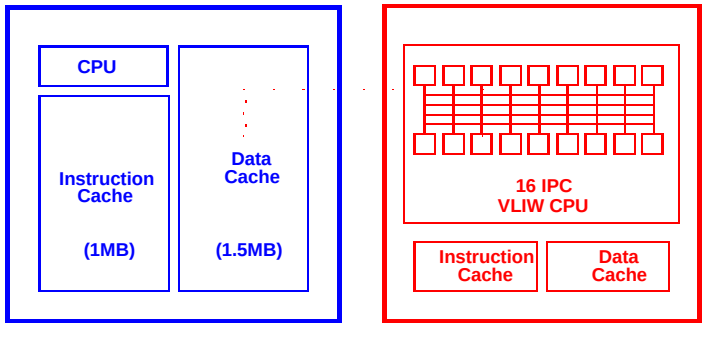
- Architecture is "*Compiler Friendly*"
  - Implementation is completely exposed - 0 layer of interpretation
  - Compile time information is easily propagated to run time
- Exceptions and interrupts are easily managed
- Run-time behavior is highly predictable
  - Allows real-time applications
  - Greater potential for code optimization

Weaknesses of VLIW Technology

- No object code compatibility between generations
- Program size is large (explicit NOPs)
  - *Multiflow machines predated "dynamic memory compression" by encoding NOPs in the instruction memory*
- Compilers are extremely complex
  - Assembly code is almost impossible
- Philosophically incompatible with caching techniques
- VLIW memory systems can be very complex
  - Simple memory systems may provide very low performance
  - Program controlled multi-layer, multi-banked memory
- Parallelism is underutilized for some algorithms

Why Renewed Interest in VLIW Now?

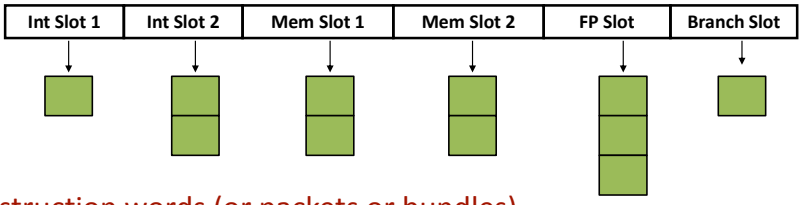
- Complexity of Scaling Superscalar Processors
  - ILP and complexity
  - Power inefficient
- Better compilation technology
  - Run-time compilation
  - Data parallel apps



1 Billion Transistor Superscalar Processor

1 Billion Transistor VLIW Processor

VLIW: Very Long Instruction Word Instruction Sets



- Long instruction words (or packets or bundles)
  - Each word contains multiple operations
- No data dependences between operations in a word (parallelism)
  - No need for RAW checks
- Each operation slot corresponds to specific functional unit
  - Operation latencies are typically fixed
- Words spaced apart statically (using nops)
  - All operations are ready to execute

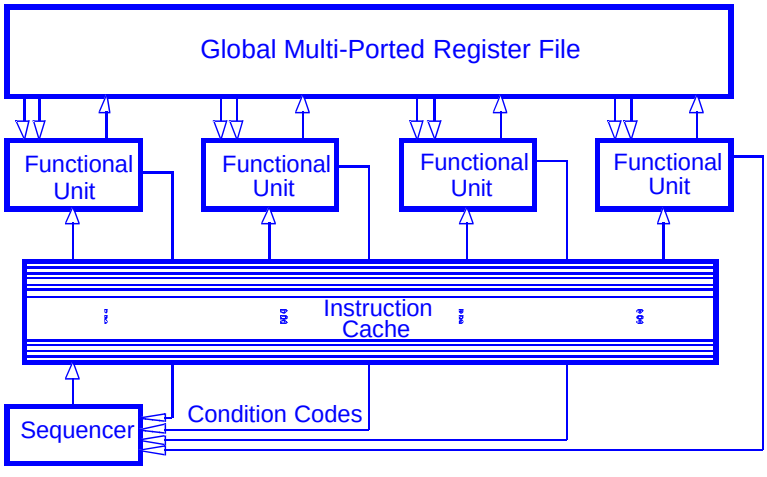
VLIW Implications

- The (optimizing) compiler must
  - Guarantee all operations within a long word are independent
    - Use nops when needed
  - Guarantee spacing between long words to avoid hazards
    - Use nops when needed
- But hardware is simple (few dynamic decisions)
  - No need for dependence checks within a word
  - No need for scheduling and issue hardware (instruction window)
  - Simple logic for functional unit assignment
- World length: 2 to 10s of instructions; 64 to 1,000 bits

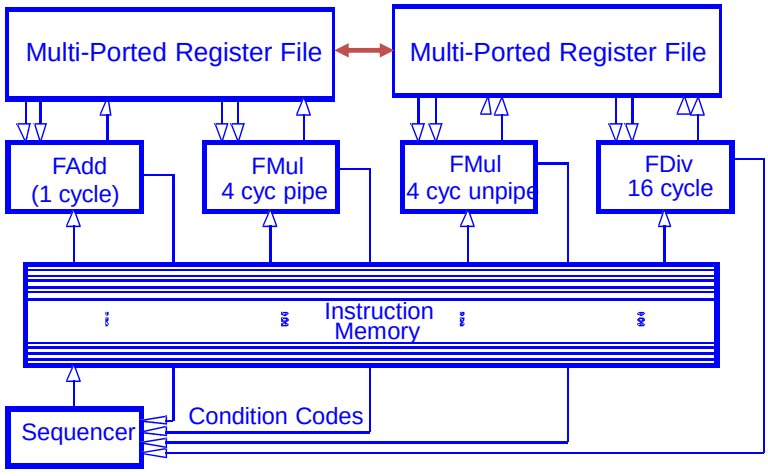
Early VLIW Models

- Josh Fisher proposed the first VLIW machine at Yale (1983)
  - Fisher's Trace Scheduling algorithm for microcode compaction could exploit more ILP than any existing processor could provide at the time
- The ELI-512 had massive resources for a single instruction stream
  - 16 processing clusters with multiple FUs per cluster
  - Partial crossbar interconnect
  - Multiple memory banks
  - Attached processor – no I/O, no operating system
- Later VLIW models became increasingly more regular
  - Compiler complexity was a greater issue than originally envisioned

Ideal VLIW Design



Realistic VLIW Design using Clustering



Clustered VLIW Architecture

- HW divided in clusters
  - Cluster: partition of registers and a few FUs
  - Full connectivity, low latency within a cluster
  - Explicit instructions to move data between clusters
- Advantage
  - Limit complexity of register file partitions
- Disadvantage
  - Compiler responsible for moving data between clusters
  - And to hide the latency of the transfers
- Clustering is common in embedded VLIW processors

Enabling Technologies for VLIW

- VLIW architectures achieve high performance through the combination of a number of key enabling hardware and software technologies.
  - **Optimizing trace scheduling** (compiler)
  - Loop unrolling & software pipelining (compiler)
  - Static branch prediction (compiler)
  - Symbolic memory disambiguation (compiler)
  - Predicated execution (compiler and HW)
  - (Software) speculative execution (compiler and HW)
  - Program compression (mostly HW)

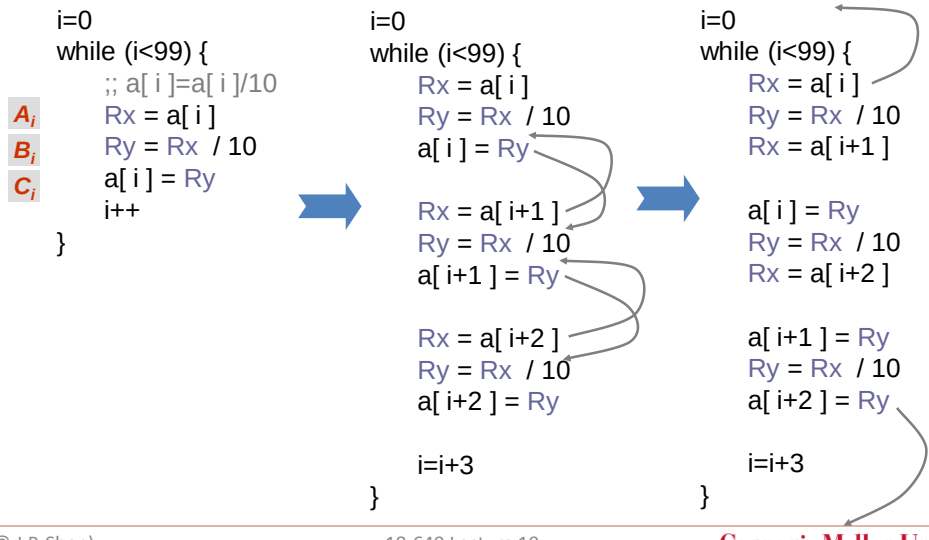
Loop Unrolling

```
i = 1;
while ( i < 100 ) {
    a[i] = b[i+1] + (i+1)/m
    b[i] = a[i-1] - i/m
    i = i + 1
}
```

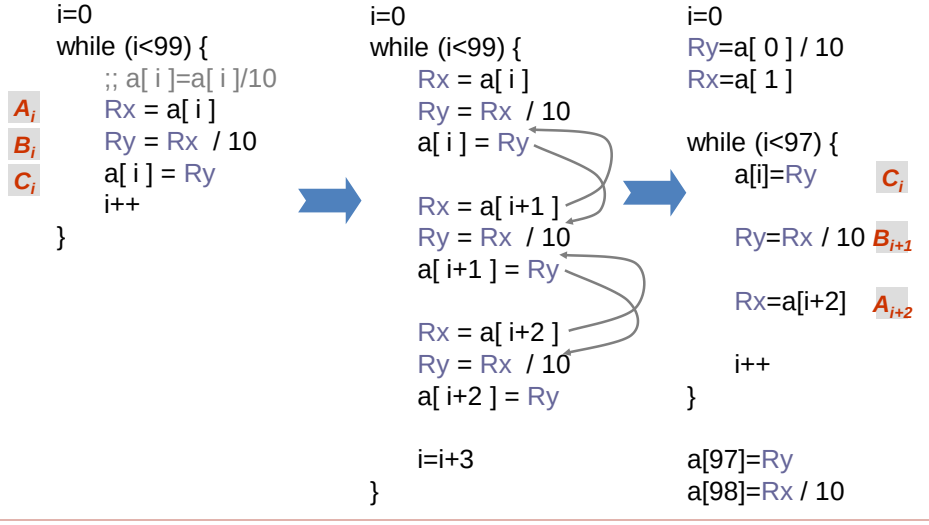
```
i = 1;
while ( i < 100 ) {
    a[i] = b[i+1] + (i+1)/m
    b[i] = a[i-1] - i/m
    a[i+1] = b[i+2] + (i+2)/m
    b[i+1] = a[i] - (i+1)/m
    i = i + 2
}
```

- Reduce loop overhead
  - Increment induction variable
  - Loop condition test
- Enlarged basic block (and analysis scope)
  - Instruction-level parallelism
  - More common subexpression
  - Memory accesses (aggressive memory aliasing analysis)

Software Pipelining

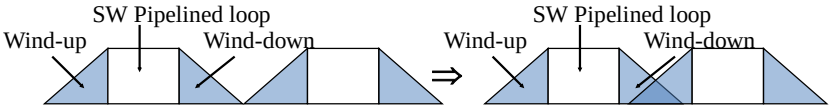


Software Pipelining *(continued)*



Software Pipelining Overhead

- SW pipelined loops require wind-up/down code
  - Wind-up code is the N iterations of the body necessary to set up the assumed register contents of the pipelined code
  - Wind-down code is the M iterations of the body necessary to finish the computations started in the pipelined code
  - For small iteration counts, the overhead may be significant



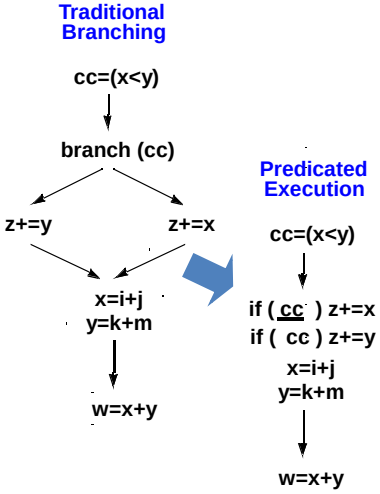
- Sometimes can overlap wind-down of one loop with wind-up of another

Predicated Execution

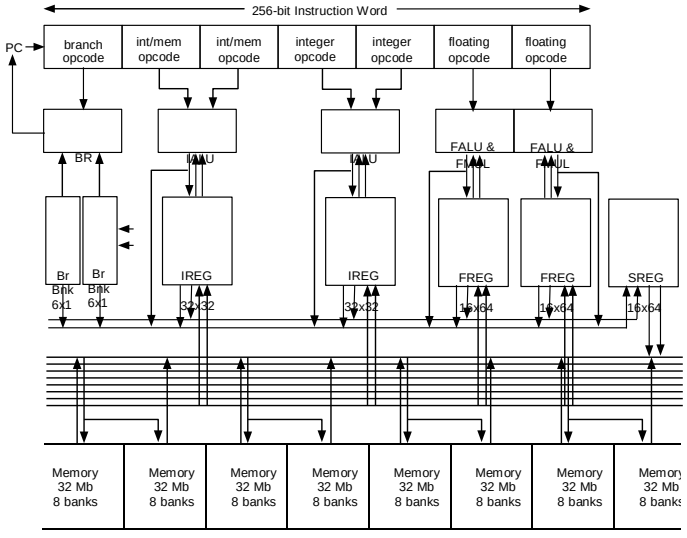
- Predicated Execution removes branches by *conditionally* executing operations
- Removing branches combines multiple basic blocks into larger basic blocks
- Branch related stalls are eliminated
- Additional opportunities for scheduling optimizations appear

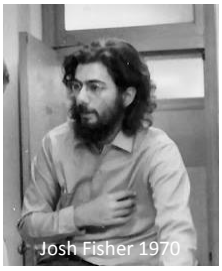
Example:

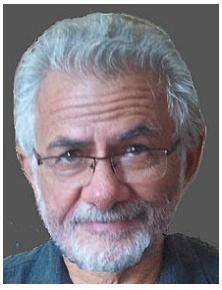
```
if (x<y) then
   z+=x;
else
   z+=y;
x=i+j;
y=k+m;
w=x+y;
```



D. The Multiflow TRACE 7/300

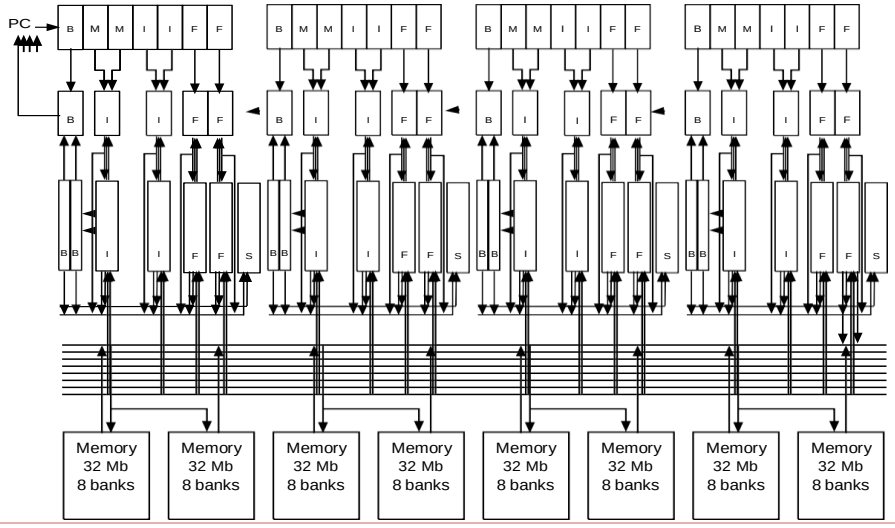




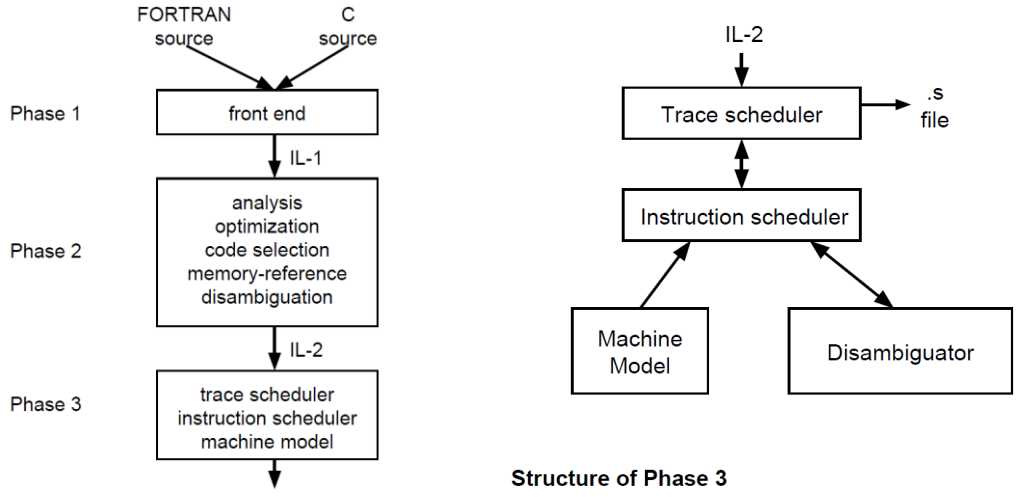


Josh Fisher 2013

The Multiflow TRACE 28/300

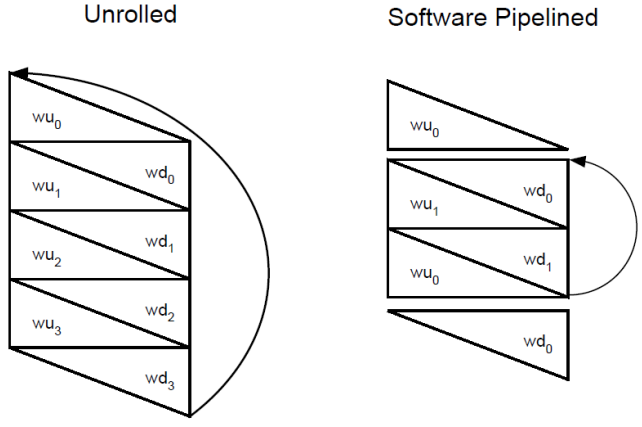


The Multiflow Compiler



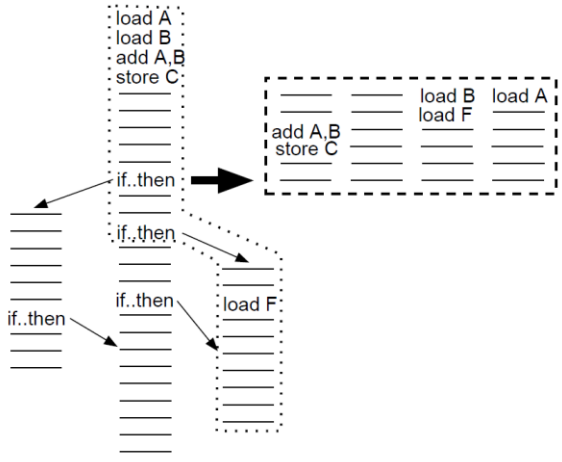
Structure of Phase 3

Loop Unrolling and Software Pipelining



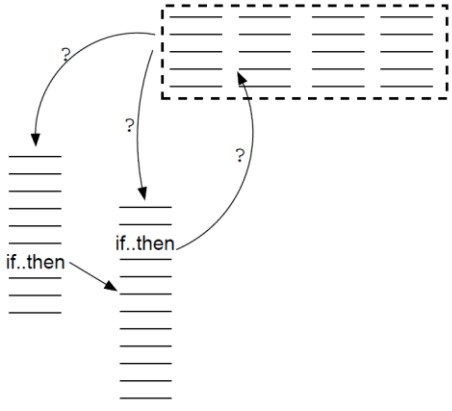
Loop unrolling and software pipelining

Multiflow Trace Scheduling



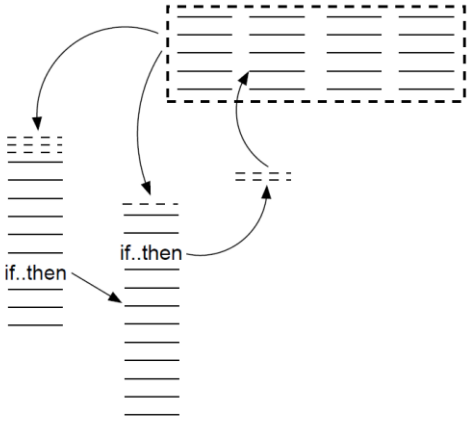
Select a trace and schedule code within the trace

Multiflow Trace Scheduling



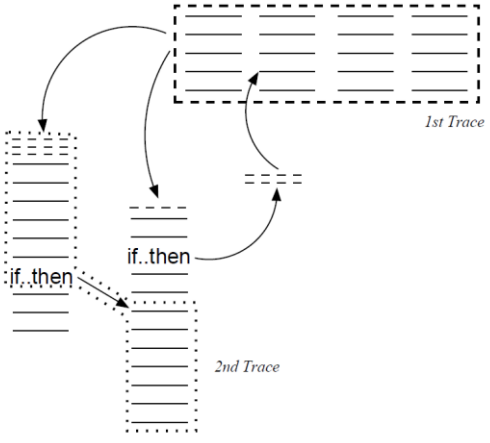
Replace the trace with the scheduled code and analyze state at split and join points

Multiflow Trace Scheduling



Generate compensation code to resolve split and join state differences

Multiflow Trace Scheduling



Iterate, selecting each trace based on successive priority

VLIW Today

- Servers: Intel IA-64 architecture
  - EPIC ISA (explicitly parallel instruction computing)
  - Chips Merced, Madison, Montecito, Tukwilla, …
  - Questionable success compared to superscalars
- Embedded: TI, NXP, ST, …
  - Very successful in low-end and high-end embedded SOCs
  - Rational: good performance/power, no need for binary compatibility
  - Large variety of ISAs, designs, optimizations points
- Interesting VLIWs of the recent past
  - Transmeta Crusoe (x86 on VLIW)

IA-64 EPIC vs. Classic VLIW

- Similarities:
  - Compiler generated wide instructions
  - Static detection of dependences
  - ILP encoded in the binary (a group)
  - Large number of architected registers
- Differences:
  - Instructions in a bundle can have dependences
  - Hardware interlock between dependent instructions
  - Accommodates varying number of functional units and latencies
  - Allows dynamic scheduling and functional unit binding

  Static scheduling are "suggestive" rather than absolute

  ⇒Code compatibility across generations

  but software won't run at top speed until it is recompiled so "shrink-wrap binary" might need to include multiple builds

E. Intel IA-64 (Itanium) Architecture

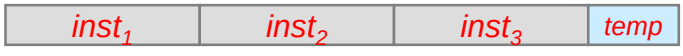
- 128 general-purpose registers
- 128 floating-point registers
- Arbitrary number of functional units
- Arbitrary latencies on the functional units
- Arbitrary number of memory ports
- Arbitrary implementation of the memory hierarchy

- *Related to but not the same as VLIW!!*
- *Binary compatible, but needs retargetable compiler and recompilation to achieve maximum program performance on different IA-64 implementations,*

IA-64 Instruction Format

- IA-64 "Bundle"
  - Total of 128 bits
  - Contains three IA-64 instructions *(aka syllables)*
  - Template bits in each bundle specify dependences both within a bundle as well as between sequential bundles
  - A collection of independent bundles forms a "group"

    A more efficient and flexible way to encode ILP then a fixed VLIW format



- IA-64 Instruction
  - Fixed-length 40 bits long
  - Contains three 7-bit register specifiers
  - Contains a 6-bit field for specifying one of the 64 one-bit predicate registers

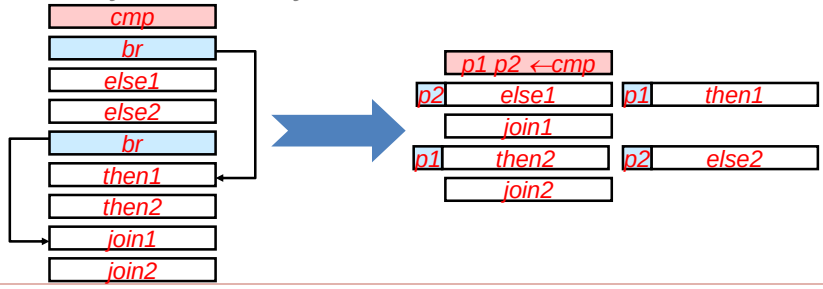
Some Cool Features of IA-64

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

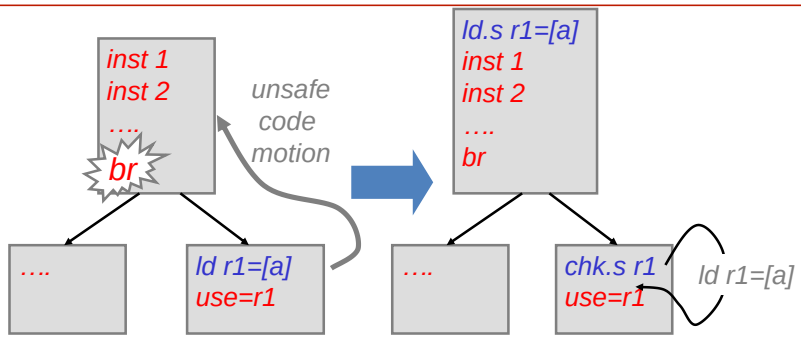
- *Mostly adapted from mechanisms that had been invented for VLIWs*

Predicated Execution

- Each instruction can be separately predicated
- 64 one-bit predicate registers
  - *Each instruction carries a 6-bit predicate field*
- An instruction is effectively a NOP if its predicate is false
- Assumes IA-64 processors have lots of spare resources
- *Converts control flow into dataflow*

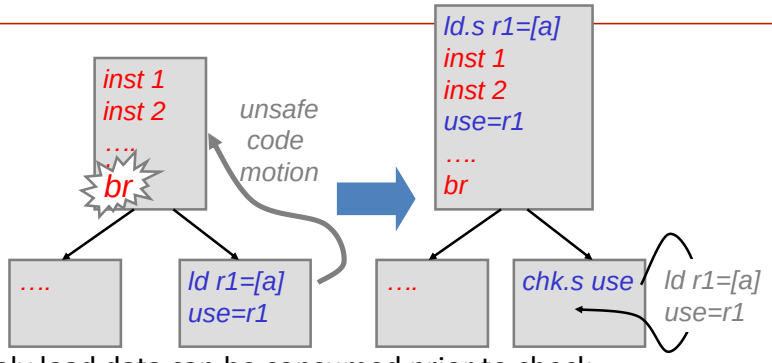


Speculative Non-Faulting Load



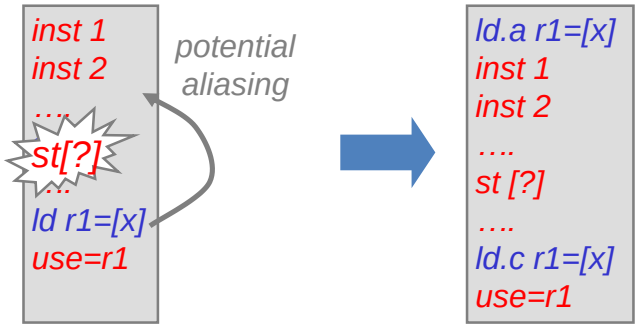
- *ld.s* fetches *speculatively* from memory
  i.e. any exception due to *ld.s* is suppressed
- If *ld.s r* did not cause an exception then *chk.s r* is an NOP, else a branch is taken (to some compensation code)

Speculative, Non-Faulting Load



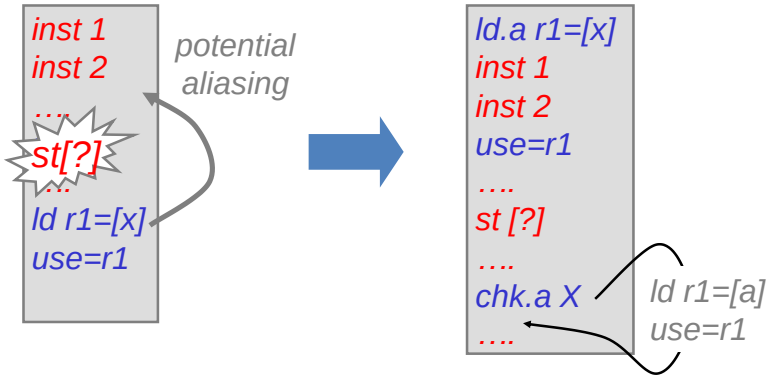
- Speculatively load data can be consumed prior to check
- "speculation" status is propagated with speculated data
- Any instruction that uses a speculative result also becomes speculative itself (i.e. suppressed exceptions)
- *chk.s* checks the entire dataflow sequence for exceptions

Speculative "Advanced" Load



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

Using Speculative Load Results



Software Assisted Branch Prediction

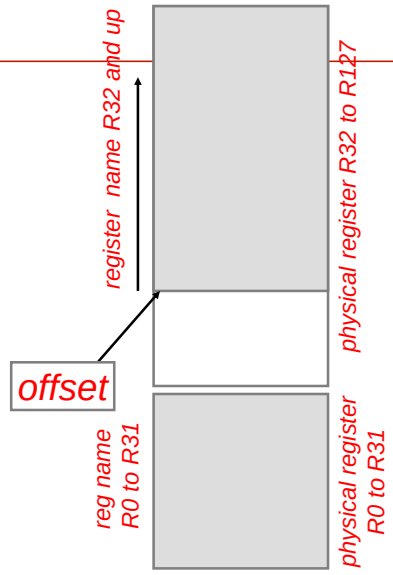
- Static branch hints can be encoded with every branch
  - Taken vs. not-taken
  - Whether to allocate an entry in the dynamic BP hardware
- Support for counted loops
  - Special register holds iteration count and used for branch prediction
- SW and HW has joint control of BP hardware
  - Original Itanium processor used a 512-entry 2-level direction predictor and 64-entry BTB
  - "brp" (branch prediction) instruction can be issued ahead of the actual branch to preset the contents of BPT and BTB
  - Brp instruction can also be used for instruction prefetching

Software Assisted Branch Prediction

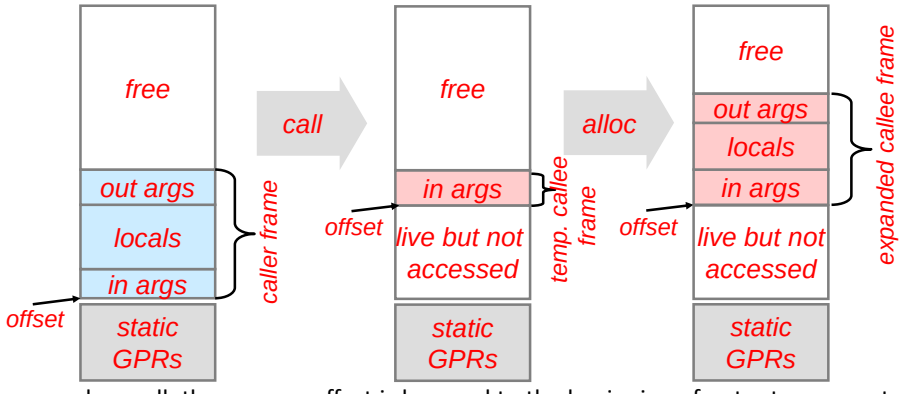
- TAR (Target Address Register)
  - a small, fully-associative BTB-like structure
  - contents are controlled entirely by a "prepare-to-branch" instruction
  - a hit in TAR overrides all other predictions
- RSB (Return Address Stack)
  - Return addr is pushed (or popped) when a procedure is called (or when it returns)
  - Predicts nPC when executing register-indirect branches

Register Renaming

- 128 general purpose physical integer registers
- Register names R0 to R31 are static and refer to the first 32 physical GPRs
- Register names R32 to R127 are known as "rotating registers" and are renamed onto the remaining 96 physical registers by an offset
- Remapping wraps around the rotating registers such that when offset is non-zero, physical location of R127 is just below R32

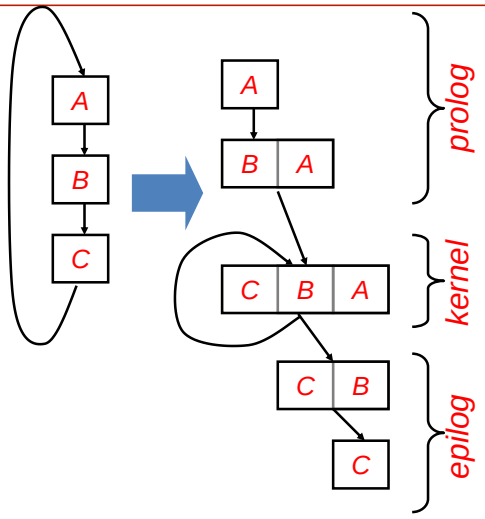


Register Stack for Procedure Calls



- On a procedure call, the rename offset is bumped to the beginning of output argument registers
- Callee can then allocate its own working frame (up to 96 regs)
- If there isn't enough free regs to be allocated, HW automatically frees up space by *spilling* live contents not in the current frame to memory *Register stack appears infinite to SW*

Rotating Loop Frames for Loop Pipelining



- Suppose B_i is only data dependent (through data stored in registers) on A_i; and C_i only on B_i
- The "pipelined" kernel block (containing independent computation from C_i, B_{i+1} and A_{i+2}) potentially has better ILP
- What happens if C_i is also data dependent on A_i
- The result placed in register by A gets clobbered by the next execution of A (in the next cycle) before C can use it two cycles from now

Nice Software Pipelining Example

```
i=0
while (i<99) {
    ;; a[ i ]=a[ i ]/10
A_i Rx = a[ i ]
B_i Ry = Rx  / 10
C_i a[ i ] = Ry
    i++
}
```

```
i=0
Ry=a[ 0 ] / 10
Rx=a[ 1 ]

while (i<97) {
    a[i]=Ry        C_i

    Ry=Rx / 10     B_{i+1}

    Rx=a[i+2]      A_{i+2}

    i++
}

a[97]=Ry
a[98]=Rx / 10
```

Software Pipelining with Renaming

```
i=0
while (i<99) {
    ;; a[ i ]=a[ i ]/10+a[ i ]
A_i Rx = a[ i ]
B_i Ry = Rx  / 10
C_i a[ i ] = Ry+Rx
    i++
}
```

```
i=0
Ry=a[ 0 ] / 10
Rx=a[ 1 ]

while (i<97) {
    a[i]=Ry+Rx      C_i
    Ry=Rx / 10      B_i+1
    Rx=a[i+2]       A_i+2
    i++
}

a[97]=Ry + Rx
a[98]=Rx / 10 + Rx
```

```
i=0
Ry=a[ 0 ] / 10
Rx=a[ 1 ]

while (i<97) {
    a[i]=Ry+Rx'
    Ry=Rx / 10
    Rx'=Rx
    Rx=a[i+2]
    i++
}

a[97]=Ry + Rx'
a[98]=Rx / 10 + Rx
```

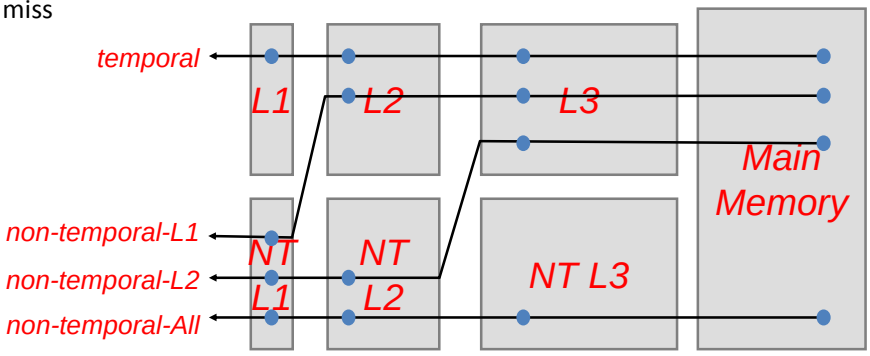

Renaming with Rotating Registers

```
i=0
Ry=a[ 0 ] / 10
Rx=a[ 1 ]

while (i<97) {
    a[i]=Ry+Rx'
    Ry=Rx / 10
    Rx'=Rx
    Rx=a[i+2]
    i++
}

a[97]=Ry + Rx'
a[98]=Rx / 10 + Rx
```

```
i= -2

while (i<99) {
    pred(i>-1):
        a[i]=Ry+RR(x-2)

    pred(i>-2 && i<98):
        Ry=RR(x-1) / 10

    pred(i<97):
        RR(x)=a[i+2]

    `increase RR offset by 1'
    i++
}
```

Software-Assisted Memory Hierarchies

- ISA provides for separate storages for "temporal" vs "non-temporal" data, each with its own multiple level of hierarchies
- Load and Store instructions can give hints about where cached copies should be held after a cache miss



Latest IA-64 Itanium Core

- 6 wide fetch & issue
  - 6 wide integer, 2 wide FP, 4 wide ld/st, 3 wide branch
  - 1 cycle L1 data cache
  - 8-stage pipeline
  - 2-threads
- Memory hierarchy
  - Separate L1I, L1D, L2I, and L2D
  - 6MB L3
  - In 4-core Tuckwilla: 24MB L3!