# CIS 501
Introduction to Computer Architecture

Unit 7: Multiple Issue and Static Scheduling

Remainder of CIS501: Parallelism

- Last unit: pipeline-level parallelism
  - Work on execute of one instruction in parallel with decode of next
- Next: instruction-level parallelism (ILP)
  - Execute multiple independent instructions fully in parallel
  - Today: limited multiple issue
  - Next week: dynamic scheduling
    - Extract much more ILP via out-of-order processing
- Data-level parallelism (DLP)
  - Single-instruction, multiple data
  - Example: one instruction, four 16-bit adds (using 64-bit registers)
- Thread-level parallelism (TLP)
  - Multiple software threads running on multiple processors

This Unit: Multiple Issue/Static Scheduling

Application	
OS	
Compiler	Firmware

| CPU	I/O
Memory	
Digital Circuits	
Gates & Transistors	
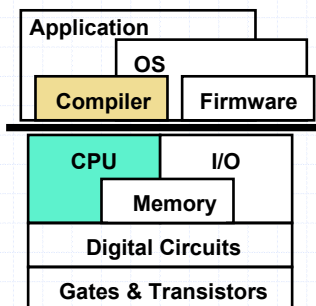
- Multiple issue scaling problems
  - Dependence-checks
  - Bypassing

- Multiple issue designs
  - Statically-scheduled superscalar
  - VLIW/EPIC (IA64)

- Advanced static scheduling

- Advanced hardware technique
  - Grid processor

Readings

- H+P
  - Chapter 4.1, 4.2, 4.3,
    4.5 ("Conditional or Predicated Instructions"),
    4.7

- Paper
  - "Superscalar Instruction Execution in the
    21164 Alpha Microprocessor" for Tuesday

Scalar Pipeline and the Flynn Bottleneck
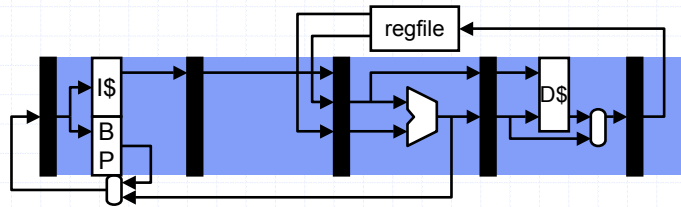
- So far we have looked at **scalar pipelines**
  - One instruction per stage
  – Performance limit (aka "Flynn Bottleneck") is CPI = IPC = 1
  – Limit is never even achieved (hazards)
  – Diminishing returns from "super-pipelining" (hazards + overhead)

Multiple-Issue Pipeline
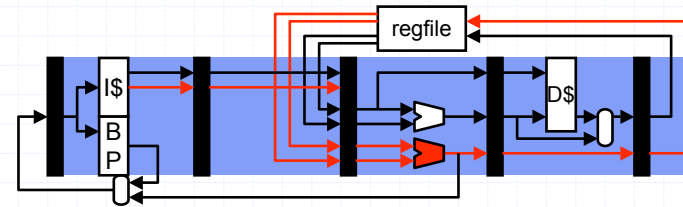
- Overcome this limit using **multiple issue**
  - Also sometimes called **superscalar**
  - Two instructions per stage at once, or three, or four, or eight…
  - "Instruction-Level Parallelism (ILP)" [Fisher]

Superscalar Execution

Single-issue	1	2	3	4	5	6	7	8	9	10	11	12
ld [r1+0]➜r2	F	D	X	M	W							
ld [r1+4]➜r3		F	D	X	M	W						
ld [r1+8]➜r4			F	D	X	M	W					
ld [r1+12]➜r5				F	D	X	M	W				
add r2,r3➜r6					F	D	X	M	W			
add r4,r6➜r7						F	D	X	M	W		
add r5,r7➜r8							F	D	X	M	W	
ld [r8]➜r9								F	D	X	M	W

| **Dual-issue**	1	2	3	4	5	6	7	8	9	10	11
ld [r1+0]➜r2	F	D	X	M	W						
ld [r1+4]➜r3	F	D	X	M	W						
ld [r1+8]➜r4		F	D	X	M	W					
ld [r1+12]➜r5		F	D	X	M	W					
add r2,r3➜r6			F	D	X	M	W				
add r4,r6➜r7			F	D	d*	X	M	W			
add r5,r7➜r8				F	D	d*	X	M	W		
ld [r8]➜r9				F	s*	D	d*	X	M	W	

Superscalar Challenges - Front End

- **Wide instruction fetch**
  - Modest: need multiple instructions per cycle
  - Aggressive: predict multiple branches, trace cache
- **Wide instruction decode**
  - Replicate decoders
- **Wide instruction issue**
  - Determine when instructions can proceed in parallel
  - Not all combinations possible
  - More complex stall logic - order N^2 for *N*-wide machine
- **Wide register read**
  - One port for each register read
  - Example, 4-wide superscalar ➜ 8 read ports

Superscalar Challenges - Back End

- **Wide instruction execution**
  - Replicate arithmetic units
  - Multiple cache ports
- **Wide instruction register writeback**
  - One write port per instruction that writes a register
  - Example, 4-wide superscalar ➜ 4 write ports
- **Wide bypass paths**
  - More possible sources for data values
  - Order ($N^2 * P$) for N-wide machine with execute pipeline depth P

- Fundamental challenge:
  - Amount of ILP (instruction-level parallelism) in the program
  - Compiler must schedule code and extract parallelism

Simple Dual-issue Pipeline
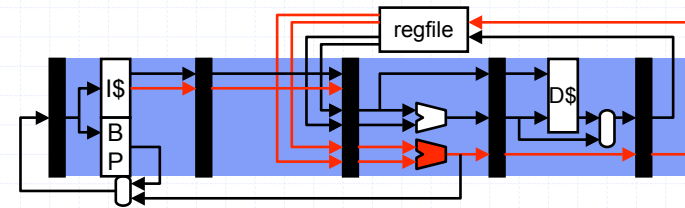
- Fetch an entire 16B or 32B cache block
  - 4 to 8 instructions (assuming 4-byte fixed length instructions)
  - Predict a single branch per cycle
- Parallel decode
  - Need to check for conflicting instructions
  - Output of I_1 is an input to I_2
  - Other stalls, too (for example, load-use delay)

Simple Dual-issue Pipeline
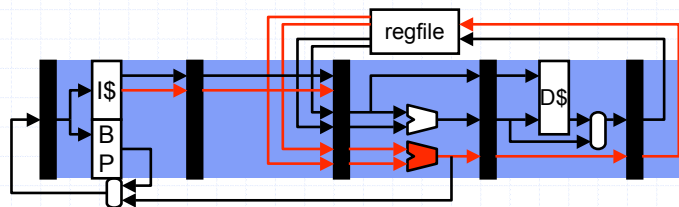
- Multi-ported register file
  - Larger area, latency, power, cost, complexity
- Multiple execution units
  - Simple adders are easy, but bypass paths are expensive
- Memory unit
  - Option #1: single load per cycle (stall at decode)
  - Option #2: add a read port to data cache
    - Larger area, latency, power, cost, complexity

Another Approach: Split Int/FP
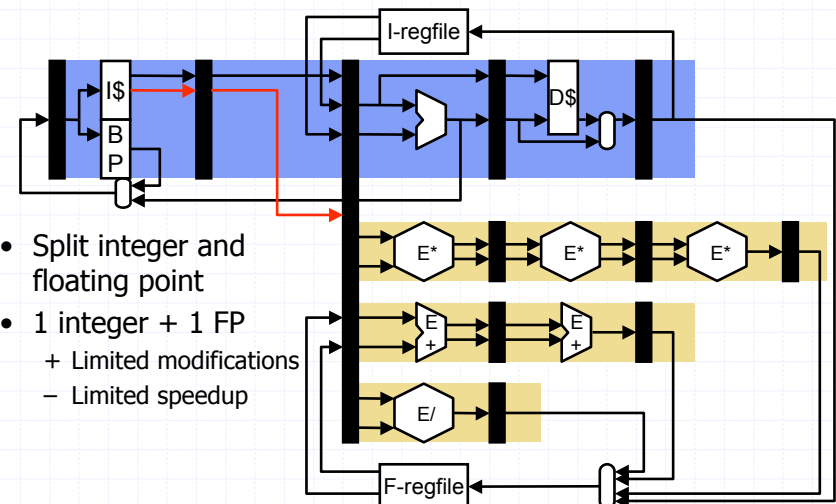
- Split integer and floating point
- 1 integer + 1 FP
  - + Limited modifications
  - – Limited speedup

Four-issue pipeline (2 integer, 2 FP)
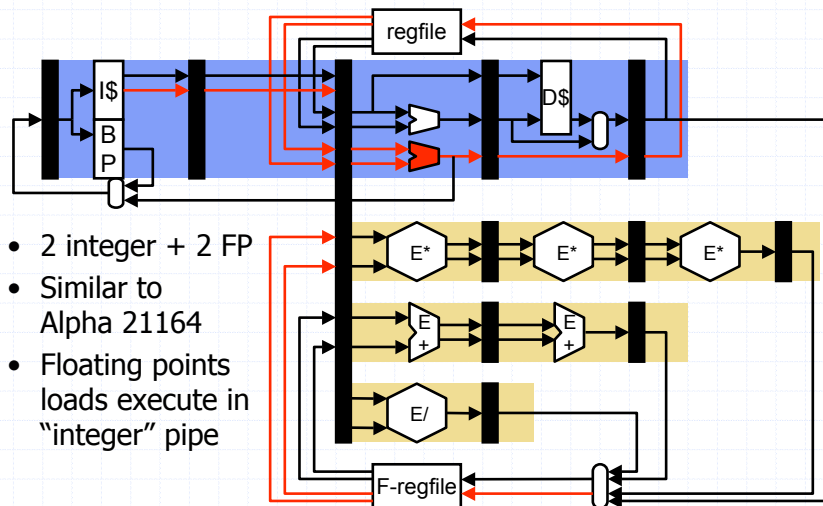
- 2 integer + 2 FP
- Similar to Alpha 21164
- Floating points loads execute in "integer" pipe

Superscalar Challenges

- Next-generation machines are 4-, 6-, 8-issue machines
- Hardware challenges
  - Wide instruction fetch
  - Wide instruction decode
  - Wide instruction issue
  - Wide register read
  - Wide instruction execution
  - Wide instruction register writeback
  - Wide bypass paths
- Extracting and exploiting available ILP
  - Hardware and software

- Let's talk about some of these issues...

Wide Fetch - Sequential Instructions
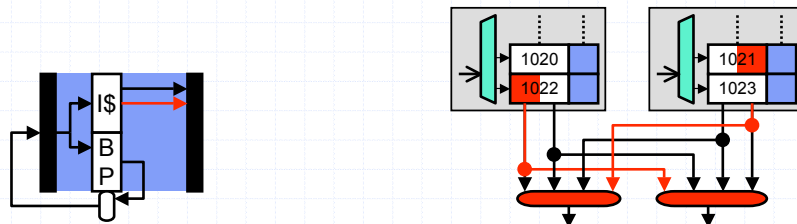
- What is involved in fetching multiple instructions per cycle?
- In same cache block? → no problem
  - Favors larger block size (independent of hit rate)
- Compilers align basic blocks to I$ lines (pad with `nops`)
  - Reduces I$ capacity
  + Increases fetch bandwidth utilization (more important)
- In multiple blocks? → Fetch block A and A+1 in parallel
  - Banked I$ + **combining network**
  - May add latency (add pipeline stages to avoid slowing down clock)

Wide Fetch - Non-sequential

- Two related questions
  - How many branches predicted per cycle?
  - Can we fetch from multiple taken branches per cycle?

- Simplest, most common organization: "1" and "No"
  - One prediction, discard post-branch insns if prediction is "Taken"
  - Lowers effective fetch width and IPC
  - Average number of instructions per taken branch?
    - Assume: 20% branches, 50% taken → ~10 instructions
  - Consider a 10-instruction loop body with an 8-issue processor
    - Without smarter fetch, ILP is limited to 5 (not 8)

- Compiler can help
  - Unroll loops, reduce taken branch frequency

Parallel Non-Sequential Fetch
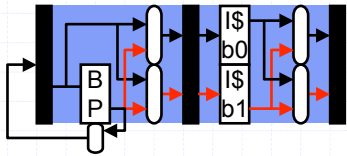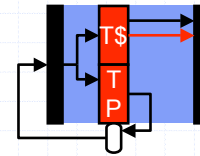

- Allowing "embedded" taken branches is possible
  - Requires smart branch predictor, multiple I$ accesses in one cycle
- Can try pipelining branch prediction and fetch
  - Branch prediction stage only needs PC
  - Transmits two PCs to fetch stage, PC and target PC
  - Elongates pipeline, increases branch penalty
  - Pentium II & III do something like this

Trace Cache

- **Trace cache (T$)** [Peleg+Weiser, Rotenberg+]
  - Overcomes serialization of prediction and fetch by combining them
  - New kind of I$ that stores **dynamic**, not static, insn sequences
    - Blocks can contain statically non-contiguous insns
    - Tag: PC of first insn + N/T of embedded branches
  - Used in Pentium 4 (actually stores decoded μops)
- Coupled with **trace predictor (TP)**
  - Predicts next trace, not next branch

Trace Cache Example

- Traditional instruction cache

Tag	Data (insns)
0	addi,beq #4,ld,sub
4	st,call #32,ld,add

	1	2
0: addi r1,4,r1	F	D
1: beq r1,#4	F	D
4: st r1,4(sp)	f*	F
5: call #32	f*	F

- Trace cache

Tag	Data (insns)
0:T	addi,beq #4,st,call #32

	1	2
0: addi r1,4,r1	F	D
1: beq r1,#4	F	D
4: st r1,4(sp)	F	D
5: call #32	F	D

- Traces can pre-decode dependence information
  - Helps fix the N^2 dependence check problem

Aside: Multiple-issue CISC

- How do we apply superscalar techniques to CISC
  - Such as x86
  - Or CISCy ugly instructions in some RISC ISAs
- Break "macro-ops" into "micro-ops"
  - Also called "μops" or "RISC-ops"
  - A typical CISCy instruction "add [r1], [r2] ➔ [r3]" becomes:
    - Load [r1] ➔ t1 (t1 is a temp. register, not visible to software)
    - Load [r2] ➔ t2
    - Add t1, t2 ➔ t3
    - Store t3➔[r3]
  - However, conversion is expensive (latency, area, power)
  - Solution: cache converted instructions in trace cache
    - Used by Pentium 4
    - Internal pipeline manipulates only these RISC-like instructions

Wide Decode
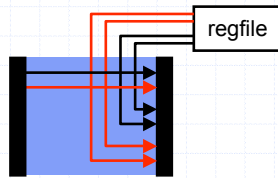
- What is involved in decoding multiple (N) insns per cycle?
- Actually doing the decoding?
  - Easy if fixed length (multiple decoders), doable if variable length
- Reading input registers?
  - – 2N register read ports (latency $\propto$ #ports)
  - + Actually less than 2N, most values come from bypasses
- What about the stall logic?

N² Dependence Cross-Check

- Stall logic for 1-wide pipeline with full bypassing
  - Full bypassing = load/use stalls only
    - X/M.op==LOAD && (D/X.rs1==X/M.rd || D/X.rs2==X/M.rd)
  - Two "terms": $\propto$ 2N
- Now: same logic for a 2-wide pipeline
  - X/M$_1$.op==LOAD && (D/X$_1$.rs1==X/M$_1$.rd || D/X$_1$.rs2==X/M$_1$.rd) ||
  - X/M$_1$.op==LOAD && (D/X$_2$.rs1==X/M$_1$.rd || D/X$_2$.rs2==X/M$_1$.rd) ||
  - X/M$_2$.op==LOAD && (D/X$_1$.rs1==X/M$_2$.rd || D/X$_1$.rs2==X/M$_2$.rd) ||
  - X/M$_2$.op==LOAD && (D/X$_2$.rs1==X/M$_2$.rd || D/X$_2$.rs2==X/M$_2$.rd)
  - Eight "terms": $\propto$ 2N²
    - This is the **N² dependence cross-check**
  - Not quite done, also need
    - D/X2.rs1==D/X1.rd || D/X2.rs2==D/X1.rd

Superscalar Stalls

- Invariant: stalls propagate upstream to younger insns
- If older insn in pair stalls, younger insns must stall too
- What if younger insn stalls?
  - Can older insn from younger group move up?
  - **Fluid**: yes, but requires some muxing
    - ± Helps CPI a little, hurts clock a little
  - **Rigid**: no
    - ± Hurts CPI a little, but doesn't impact clock

Rigid	1	2	3	4	5		Fluid	1	2	3	4	5
ld 0(r1),r4	F	D	X	M	W		ld 0(r1),r4	F	D	X	M	W
addi r4,1,r4	F	D	d*	d*	X		addi r4,1,r4	F	D	d*	d*	X
sub r5,r2,r3		F	p*	p*	D		sub r5,r2,r3		F	D	p*	X
st r3,0(r1)		F	p*	p*	D		st r3,0(r1)		F	p*	p*	D
ld 4(r1),r8					F		ld 4(41),r8				F	p*	D

Wide Execute
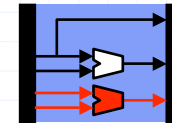

- What is involved in executing multiple (N) insns per cycle?
- Multiple execution units … N of every kind?
  - N ALUs? OK, ALUs are small
  - N FP dividers? No, FP dividers are huge and `fdiv` is uncommon
  - How many branches per cycle?
  - How many loads/stores per cycle?
  - Typically some mix of functional units proportional to insn mix
    - Intel Pentium: 1 any + 1 ALU

Wide Memory Access

- How do we allow multiple loads/stores to execute?
  - Option#1: Extra read ports on data cache
    - Higher latency, etc.
  - Option#2: "Bank" the cache
    - Can support a load to an "odd" and an "even" address
    - Problem: address not known to execute stage
      - Complicates stall logic
      - With two banks, conflicts will occur frequently
  - Option #3: Replicate the cache
    - Multiple read bandwidth only
    - Larger area, but no conflicts, can be faster than more ports
    - Independent reads to replicas, writes (stores) go to all replicas
- Example: the Alpha 21164
  - 8KB L1-caches, supports two loads, but only one store
  - They probably use option #3

N^2 Bypass Network
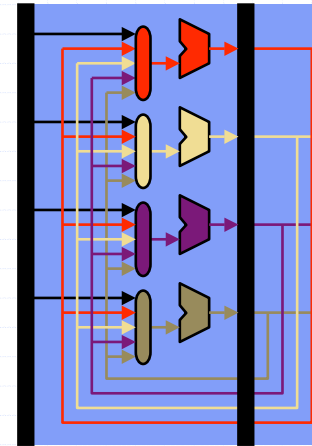
- N^2 stall and bypass logic
  - Actually OK
  - 5-bit and 1-bit quantities
- **N^2 bypass network**
  - 32-bit (or 64-bit) quantities
  - Routing lengthens wires
  - Expensive metal layer crossings
  - N+1 input muxes at each ALU input
  - And this is just one bypassing stage!
- Bit-slicing
  - Mitigates routing problem somewhat
  - 32 or 64 1-bit bypass networks

Clustering
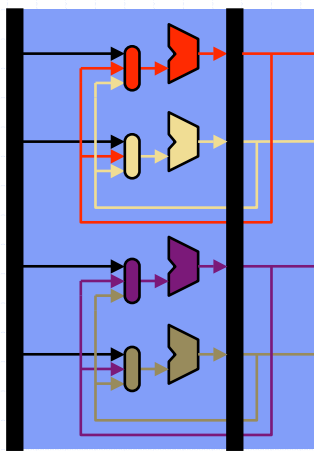
- **Clustering**: mitigates N^2 bypass
  - Group FUs into **K** clusters
  - Full bypassing within a cluster
  - Limited bypassing between clusters
    - With a one cycle delay
  - (N/K) + 1 inputs at each mux
  - $(N/K)^2$ bypass paths in each cluster
- **Steering**: key to performance
  - Steer dependent insns to same cluster
  - Statically (compiler) or dynamically
- E.g., Alpha 21264
  - Bypass wouldn't fit into clock cycle
  - 4-wide, 2 clusters, static steering
  - Replicates register file, too

Wide Writeback
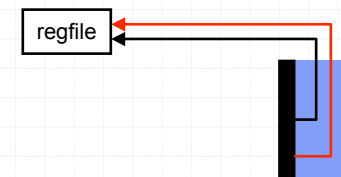
- What is involved in multiple (N) writebacks per cycle?
  - N register file write ports (latency $\propto$ #ports)
  - Usually less than N, stores and branches don't do writeback
- Multiple exceptions per cycle?
  - No just the oldest one

Multiple-Issue Implementations

- **Statically-scheduled (in-order) superscalar**
  - \+ Executes unmodified sequential programs
  - – Hardware must figure out what can be done in parallel
  - E.g., Pentium (2-wide), UltraSPARC (4-wide), Alpha 21164 (4-wide)
- **Very Long Instruction Word (VLIW)**
  - \+ Hardware can be dumb and low power
  - – Compiler must group parallel insns, requires new binaries
  - E.g., TransMeta Crusoe (4-wide)
- **Explicitly Parallel Instruction Computing (EPIC)**
  - A compromise: compiler does some, hardware does the rest
  - E.g., Intel Itanium (6-wide)
- **Dynamically-scheduled superscalar**
  - Pentium Pro/II/III (3-wide), Alpha 21264 (4-wide)
- We'll already talked about statically-scheduled superscalar

VLIW

- Hardware-centric multiple issue problems
  - – Wide fetch+branch prediction, N^2 bypass, N^2 dependence checks
  - – Hardware solutions have been proposed: clustering, trace cache

- Software-centric: **very long insn word (VLIW)**
  - Effectively, a 1-wide pipeline, but unit is an N-insn group
  - Compiler guarantees insns within a VLIW group are independent
    - If no independent insns, slots filled with `nops`
  - Group travels down pipeline as a unit
    - \+ Simplifies pipeline control (no rigid vs. fluid business)
    - \+ Cross-checks within a group un-necessary
    - Downstream cross-checks still necessary
  - Typically "slotted": 1st insn must be ALU, 2nd mem, etc.
    - \+ Further simplification

History of VLIW

- Started with "horizontal microcode"
- Academic projects
  - Yale ELI-512 [Fisher, '85]
  - Illinois IMPACT [Hwu, '91]
- Commercial attempts
  - Multiflow [Colwell+Fisher, '85] → failed
  - Cydrome [Rau, '85] → failed
  - Motorolla/TI embedded processors → successful
  - Intel Itanium [Colwell,Fisher+Rau, '97] → ??
  - Transmeta Crusoe [Ditzel, '99] → mostly failed

Pure and Tainted VLIW

- **Pure VLIW**: no hardware dependence checks at all
  - Not even between VLIW groups
  - \+ Very simple and low power hardware
  - Compiler responsible for scheduling stall cycles
  - Requires precise knowledge of pipeline depth and structure
    - – These must be fixed for compatibility
  - – Doesn't support caches well
  - Used in some cache-less micro-controllers, but not generally useful

- **Tainted (more realistic) VLIW**: inter-group checks
  - Compiler doesn't schedule stall cycles
  - \+ Precise pipeline depth and latencies not needed, can be changed
  - \+ Supports caches
  - TransMeta Crusoe

What Does VLIW Actually Buy Us?

+ Simpler I$/branch prediction
  - No trace cache necessary
+ Slightly simpler dependence check logic
- Bypasses are the same
  - Clustering can help VLIW, too
- Not compatible across machines of different widths
  - Is non-compatibility worth all of this?

- PS how does TransMeta deal with compatibility problem?
  - Dynamically translates x86 to internal VLIW

EPIC

- Tainted VLIW
  - Compatible across pipeline depths
  - But not across pipeline widths and slot structures
  - Must re-compile if going from 4-wide to 8-wide
  - TransMeta sidesteps this problem by re-compiling transparently

- **EPIC (Explicitly Parallel Insn Computing)**
  - New VLIW (Variable Length Insn Words)
  - Implemented as "bundles" with explicit dependence bits
  - Code is compatible with different "bundle" width machines
  - Compiler discovers as much parallelism as it can, hardware does rest
  - E.g., Intel Itanium (IA-64)
    - 128-bit bundles (3 41-bit insns + 4 dependence bits)

ILP and Static Scheduling

- No point to having an N-wide pipeline…
- …if average number of parallel insns per cycle (ILP) << N

- How can the compiler help extract parallelism?
  - These techniques applicable to regular superscalar
  - These techniques critical for VLIW/EPIC

Code Example: SAXPY

- **SAXPY** (Single-precision A X Plus Y)
  - Linear algebra routine (used in solving systems of equations)
  - Part of early "Livermore Loops" benchmark suite

```
for (i=0;i<N;i++)
  Z[i]=A*X[i]+Y[i];


0: ldf X(r1),f1       // loop
1: mulf f0,f1,f2      // A in f0
2: ldf Y(r1),f3       // X,Y,Z are constant addresses
3: addf f2,f3,f4
4: stf f4,Z(r1)
5: addi r1,4,r1       // i in r1
6: blt r1,r2,0        // N*4 in r2
```

SAXPY Performance and Utilization

```
                1  2  3  4  5 | 6  7  8  9 10 11 12 13 14 15 16|17 18 19 20
ldf X(r1),f1    F  D  X  M  W
mulf f0,f1,f2      F  D d* E*  E* E* E* E* W
ldf Y(r1),f3          F p*  D   X  M  W
addf f2,f3,f4            F  D  d* d* d* E+ E+  W
stf f4,Z(r1)               F  p* p* p*  D  X  M  W
addi r1,4,r1                             F  D  X  M  W
blt r1,r2,0                                 F  D  X  M  W
ldf X(r1),f1                                   F  D  X  M  W
```

- Scalar pipeline
  - Full bypassing, 5-cycle E*, 2-cycle E+, branches predicted taken
  - Single iteration (7 insns) latency: 16–5 = 11 cycles
  - **Performance**: 7 insns / 11 cycles = 0.64 IPC
  - **Utilization**: 0.64 actual IPC / 1 peak IPC = 64%

SAXPY Performance and Utilization

```
                1  2  3  4  5 | 6  7  8  9 10 11 12 13 14 15|16 17 18 19 20
ldf X(r1),f1    F  D  X  M  W
mulf f0,f1,f2   F  D d* d* E*  E* E* E* E* W
ldf Y(r1),f3       F  D p*  X   M  W
addf f2,f3,f4      F p* p*  D  d* d* d* d* E+ E+  W
stf f4,Z(r1)          F p*  D  p* p* p* p* d*  X  M  W
addi r1,4,r1            F  p* p* p* p* p*  D  X  M  W
blt r1,r2,0            F  p* p* p* p* p*  D d*  X  M  W
ldf X(r1),f1                              F  D  X  M  W
```

- Dual issue pipeline (fluid)
  - Same + any two insns per cycle + embedded taken branches
  + **Performance**: 7 insns / 10 cycles = 0.70 IPC
  – **Utilization**: 0.70 actual IPC / 2 peak IPC = 35%
  – More hazards → more stalls (why?)
  – Each stall is more expensive (why?)

Schedule and Issue

- **Issue**: time at which insns execute
  - Want to maintain issue rate of N
- **Schedule**: order in which insns execute
  - In in-order pipeline, schedule + stalls determine issue
  - A good schedule that minimizes stalls is important
    - For both performance and utilization

- Schedule/issue combinations
  - Pure VLIW: static schedule, static issue
  - Tainted VLIW: static schedule, partly dynamic issue
  - Superscalar, EPIC: static schedule, dynamic issue

Instruction Scheduling

- Idea: place independent insns between slow ops and uses
  - Otherwise, pipeline stalls while waiting for RAW hazards to resolve
  - Have already seen pipeline scheduling
- To schedule well need … **independent insns**
- **Scheduling scope**: code region we are scheduling
  - The bigger the better (more independent insns to choose from)
  - Once scope is defined, schedule is pretty obvious
  - Trick is creating a large scope (must schedule across branches)
- Compiler scheduling (really scope enlarging) techniques
  - Loop unrolling (for loops)
  - Trace scheduling (for non-loop control flow)

Aside: Profiling

- **Profile**: statistical information about program tendencies
  - Software's answer to everything
  - Collected from previous program runs (different inputs)
  - ± Works OK depending on information
    - Memory latencies (cache misses)
      - + Identities of frequently missing loads stable across inputs
      - – But are tied to cache configuration
    - Memory dependences
      - + Stable across inputs
      - – But exploiting this information is hard (need hw help)
    - Branch outcomes
      - – Not so stable across inputs
  - – More difficult to use, need to run program and then re-compile
  - Popular research topic

Loop Unrolling SAXPY

- Goal: separate dependent insns from one another
- SAXPY problem: not enough flexibility within one iteration
  - Longest chain of insns is 9 cycles
    - Load (1)
    - Forward to multiply (5)
    - Forward to add (2)
    - Forward to store (1)
  - – Can't hide a 9-cycle chain using only 7 insns
  - But how about two 9-cycle chains using 14 insns?
- **Loop unrolling**: schedule two or more iterations together
  - Fuse iterations
  - Pipeline schedule to reduce RAW stalls
  - Pipeline schedule introduces WAR violations, rename registers to fix

Unrolling SAXPY I: Fuse Iterations

- Combine two (in general K) iterations of loop
  - Fuse loop control: induction variable (`i`) increment + branch
  - Adjust implicit induction uses

```
ldf X(r1),f1            ldf X(r1),f1
mulf f0,f1,f2           mulf f0,f1,f2
ldf Y(r1),f3            ldf Y(r1),f3
addf f2,f3,f4           addf f2,f3,f4
stf f4,Z(r1)            stf f4,Z(r1)
addi r1,4,r1
blt r1,r2,0
ldf X(r1),f1            ldf X+4(r1),f1
mulf f0,f1,f2           mulf f0,f1,f2
ldf Y(r1),f3            ldf Y+4(r1),f3
addf f2,f3,f4           addf f2,f3,f4
stf f4,Z(r1)            stf f4,Z+4(r1)
addi r1,4,r1            addi r1,8,r1
blt r1,r2,0            blt r1,r2,0
```

Unrolling SAXPY II: Pipeline Schedule

- Pipeline schedule to reduce RAW stalls
  - Have already seen this: pipeline scheduling

```
ldf X(r1),f1            ldf X(r1),f1
mulf f0,f1,f2           ldf X+4(r1),f1
ldf Y(r1),f3            mulf f0,f1,f2
addf f2,f3,f4           mulf f0,f1,f2
stf f4,Z(r1)            ldf Y(r1),f3
ldf X+4(r1),f1         ldf Y+4(r1),f3
mulf f0,f1,f2           addf f2,f3,f4
ldf Y+4(r1),f3         addf f2,f3,f4
addf f2,f3,f4           stf f4,Z(r1)
stf f4,Z+4(r1)         stf f4,Z+4(r1)
addi r1,8,r1           addi r1,8,r1
blt r1,r2,0            blt r1,r2,0
```

Unrolling SAXPY III: Rename Registers

- Pipeline scheduling causes WAR violations
  - Rename registers to correct

```
ldf X(r1),f1          ldf X(r1),f1
ldf X+4(r1),f1        ldf X+4(r1),f5
mulf f0,f1,f2         mulf f0,f1,f2
mulf f0,f1,f2         mulf f0,f5,f6
ldf Y(r1),f3          ldf Y(r1),f3
ldf Y+4(r1),f3        ldf Y+4(r1),f7
addf f2,f3,f4         addf f2,f3,f4
addf f2,f3,f4         addf f6,f7,f8
stf f4,Z(r1)          stf f4,Z(r1)
stf f4,Z+4(r1)        stf f8,Z+4(r1)
addi r1,8,r1          addi r1,8,r1
blt r1,r2,0           blt r1,r2,0
```

Unrolled SAXPY Performance/Utilization

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
ldf X(r1),f1	F	D	X	M	W															
ldf X+4(r1),f5		F	D	X	M	W														
mulf f0,f1,f2			F	D	E*	E*	E*	E*	E*	W										
mulf f0,f5,f6				F	D	E*	E*	E*	E*	E*	W									
ldf Y(r1),f3					F	D	X	M	W											
ldf Y+4(r1),f7						F	D	X	M	s*	s*	W								
addf f2,f3,f4							F	D	d*	E+	E+	s*	W							
addf f6,f7,f8							F	p*	D	E+	p*	E+	W							
stf f4,Z(r1)									F	D	X	M	W							
stf f8,Z+4(r1)										F	D	X	M	W						
addi r1,8,r1											F	D	X	M	W					
blt r1,r2,0												F	D	X	M	W				
ldf X(r1),f1													F	D	X	M	W			

No propagation? Different pipelines

- + Performance: 12 insn / 13 cycles = 0.92 IPC
- + Utilization: 0.92 actual IPC / 1 peak IPC = 92%
- + **Speedup**: (2 * 11 cycles) / 13 cycles = 1.69

Loop Unrolling Shortcomings

- Static code growth more I$ misses
  - Limits practical unrolling limit
- Poor scheduling along "seams" of unrolled copies
- Need more registers to resolve WAR hazards
- **Doesn't handle recurrences** (inter-iteration dependences)
  - Handled by software pipelining (not further discussed)

Beyond Scheduling Loops

- Problem: not everything is a loop
  - How to create large scheduling scopes from non-loop code?
- Idea: **trace scheduling** [Ellis, '85]
  - Find common paths in program (profile)
  - Realign basic blocks to form straight-line "traces"
    - **Basic-block**: single-entry, single-exit insn sequence
    - **Trace**: fused basic block sequence
  - Schedule insns within a trace
    - This is the easy part
  - Create **fixup code** outside trace
    - In case implicit trace path doesn't equal actual path
    - Nasty
  - Good scheduling needs **ISA support for software speculation**
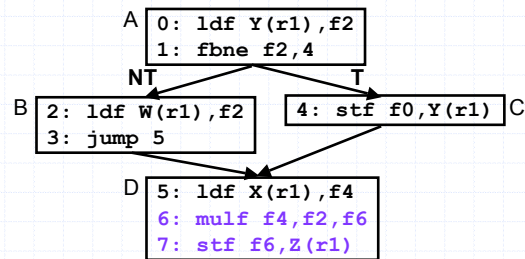
Trace Scheduling Example

Source code
```
A = Y[i];
if (A == 0)
   A = W[i];
else
   Y[i] = 0;
Z[i] = A*X[i];
```

Machine code
```
0: ldf Y(r1),f2
1: fbne f2,4
2: ldf W(r1),f2
3: jump 5
4: stf f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

4 basic blocks: A,B,C,D

A
```
0: ldf Y(r1),f2
1: fbne f2,4
```
NT   T

B
```
2: ldf W(r1),f2
3: jump 5
```
C
```
4: stf f0,Y(r1)
```

D
```
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

- Problem: separate #6 (3 cycles) from #7
- How to move `mulf` above if-then-else?
- How to move `ldf`?

Superblocks

A
```
0: ldf Y(r1),f2
1: fbne f2,4
```
NT=5%   T=95%

B
```
2: ldf W(r1),f2
3: jump 5
```
C
```
4: stf f0,Y(r1)
```

D
```
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```
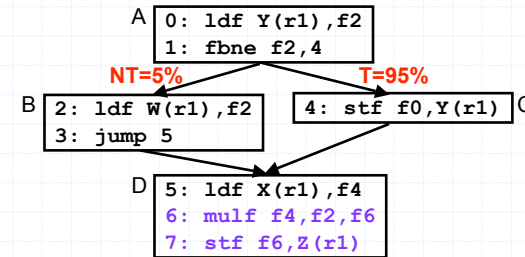
- First trace scheduling construct: **superblock**
  - Use when branch is highly biased
  - Fuse blocks from most frequent path: A,C,D
  - Schedule
  - Create **repair code** in case real path was A,B,D

Superblock and Repair Code

Superblock
```
0: ldf Y(r1),f2
1: fbeq f2,2
4: stf f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

Repair code
```
2: ldf W(r1),f2
5': ldf X(r1),f4
6': mulf f4,f2,f6
7': stf f6,Z(r1)
```

- What did we do?
  - Change sense (test) of branch 1
    - Original taken target now fall-thru
  - Created repair block
    - May need to duplicate some code (here basic-block D)
  - Haven't actually scheduled superblock yet

Superblocks Scheduling I

Superblock
```
0: ldf Y(r1),f2
1: fbeq f2,2
5: ldf X(r1),f4
6: mulf f4,f2,f6
4: stf f0,Y(r1)
7: stf f6,Z(r1)
```

Repair code
```
2: ldf W(r1),f2
5': ldf X(r1),f4
6': mulf f4,f2,f6
7': stf f6,Z(r1)
```

- First scheduling move: move insns 5 and 6 above insn 4
  - Hmmm: moved load (5) above store (4)
  - We can tell this is OK, but can the compiler
    - If yes, fine
    - Otherwise, need to do something

ISA Support for Load/Store Speculation

Superblock
```
0: ldf Y(r1),f2
1: fbeq f2,2
5: ldf.a X(r1),f4
6: mulf f4,f2,f6
4: stf f0,Y(r1)
8: chk.a f4,9
7: stf f6,Z(r1)
```

Repair code
```
2: ldf W(r1),f2
5': ldf X(r1),f4
6': mulf f4,f2,f6
7': stf f6,Z(r1)
```

Repair code 2

- IA-64: change insn 5 to **advanced load `ldf.a`**
  - "Advanced" means advanced past some unknown store
  - Processor stores [address, reg] of advanced loads in table
    - Memory Conflict Buffer (MCB), Advanced Load Alias Table (ALAT)
  - Later stores search ALAT: matching address → invalidate register
  - Insert check insn `chk.a` to make sure register is still good
  - If not, jump to some more repair code (arghhh…)

Superblock Scheduling II

Superblock
```
0: ldf Y(r1),f2
5: ldf.a X(r1),f4
6: mulf f4,f2,f6
1: fbeq f2,2
4: stf f0,Y(r1)
8: chk.a f4,9
7: stf f6,Z(r1)
```
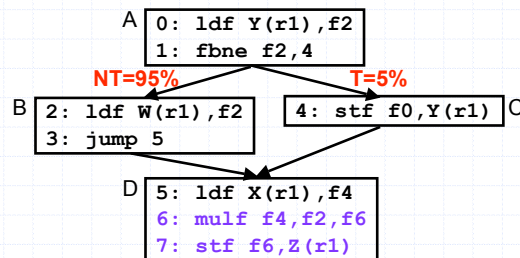
Repair code
```
2: ldf W(r1),f2
5': ldf X(r1),f4
6': mulf f4,f2,f6
7': stf f6,Z(r1)
```

Repair code 2

- Second scheduling move: move insn 5 and 6 above insn 1
  - That's OK, load did not depend on branch…
  - And would have executed anyway
- Scheduling non-move: don't move insn 4 above insn 1
  - Why? Hard (but possible) to undo a store in repair code

- **Success**: scheduled 3 insns between 6 and 7

What If…

A
```
0: ldf Y(r1),f2
1: fbne f2,4
```

NT=95%   T=5%

B
```
2: ldf W(r1),f2
3: jump 5
```

C
```
4: stf f0,Y(r1)
```

D
```
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

- … branch 1 had the opposite bias?

The Other Superblock and Repair Code

Superblock
```
0: ldf Y(r1),f2
1: fbne f2,4
2: ldf W(r1),f2
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

Repair code
```
4: stf f0,Y(r1)
5': ldf X(r1),f4
6': mulf f4,f2,f6
7': stf f6,Z(r1)
```

- Notice
  - Branch 1 sense (test) unchanged
    - Original taken target now in repair code

Superblock Scheduling III

Superblock
```
0: ldf Y(r1),f2
2: ldf W(r1),f8
5: ldf X(r1),f4
6: mulf f4,f8,f6
1: fbne f2,4
7: stf f6,Z(r1)
```

Repair code
```
4: stf f0,Y(r1)
6': mulf f4,f2,f6
7': stf f6,Z(r1)
```

- First scheduling move: move insns 2, 5, and 6 above insn 1
  - Rename `f2` to `f8` to avoid WAR violation
  - Notice, can remove copy of insn 5 from repair code
  - Is this scheduling move legal?
    - From a store standpoint, yes
    - What about from a fault standpoint? What if insn 2 faults?

ISA Support for Load-Branch Speculation

Superblock
```
0: ldf Y(r1),f2
2: ldf.s W(r1),f8
5: ldf X(r1),f4
6: mulf f4,f8,f6
1: fbne f2,4
8: chk.s f8
7: stf f6,Z(r1)
```
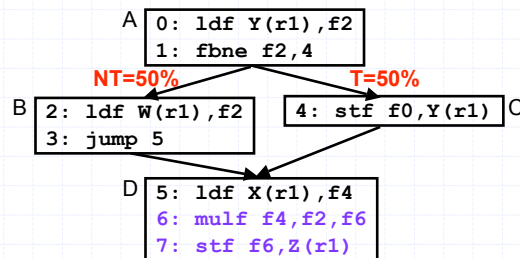
Repair code
```
4: stf f0,Y(r1)
6': mulf f4,f2,f6
7': stf f6,Z(r1)
```

Repair code 2

- IA-64: change insn 2 to **speculative load `ldf.s`**
  - "Speculative" means advanced past some unknown branch
  - Processor keeps exception bit with register `f8`
  - Inserted insn `chk.s` checks exception bit
  - If exception, jump to yet more repair code (arghhh…)

- IA-64 also contains `ldf.sa`

Non-Biased Branches: Use Predication

A
```
0: ldf Y(r1),f2
1: fbne f2,4
```

NT=50%   T=50%

B
```
2: ldf W(r1),f2
3: jump 5
```

C
```
4: stf f0,Y(r1)
```

D
```
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

↓

Using Predication
```
0: ldf Y(r1),f2
1: fspne f2,p1
2: ldf.p p1,W(r1),f2
4: stf.np p1,f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

Predication

- Conventional control
  - Conditionally executed insns also conditionally fetched
- **Predication**
  - Conditionally executed insns unconditionally fetched
  - **Full predication** (ARM, IA-64)
    - Can tag every insn with predicate, but extra bits in instruction
  - **Conditional moves** (Alpha, IA-32)
    - Construct appearance of full predication from one primitive
      ```
      cmoveq r1,r2,r3        // if (r1==0) r3=r2;
      ```
    - May require some code duplication to achieve desired effect
    + Only good way of adding predication to an existing ISA
- **If-conversion**: replacing control with predication
  + Good if branch is unpredictable (save mis-prediction)
 - But more instructions fetched and "executed"

ISA Support for Predication

```
0: ldf Y(r1),f2
1: fspne f2,p1
2: ldf.p p1,W(r1),f2
4: stf.np p1,f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

- IA-64: change branch 1 to **set-predicate insn `fspne`**
- Change insns 2 and 4 to **predicated insns**
  - `ldf.p` performs `ldf` if predicate `p1` is true
  - `stf.np` performs `stf` if predicate `p1` is false
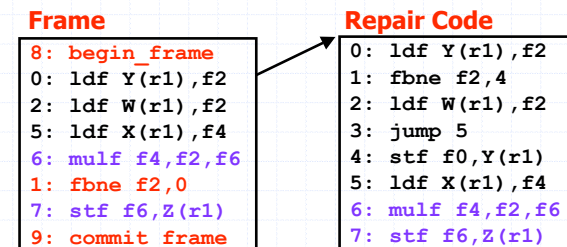
Hyperblock Scheduling

```
A  0: ldf Y(r1),f2
   1: fbne f2,4
```
NT=50%   T=50%
```
B  2: ldf W(r1),f2       4: stf f0,Y(r1)  C
   3: jump 5
```
```
D  5: ldf X(r1),f4
   6: mulf f4,f2,f6
   7: stf f6,Z(r1)
```

```
0: ldf Y(r1),f2
1: fspne f2,p1
2: ldf.p p1,W(r1),f2
4: stf.np p1,f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```
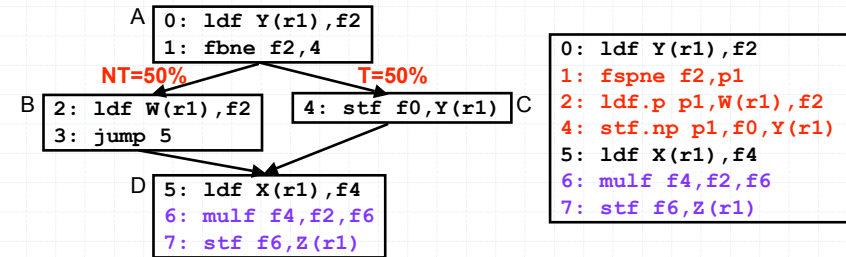
- Second trace scheduling construct: **hyperblock**
  - Use when branch is not highly biased
  - Fuse all four blocks: A,B,C,D
  - Use **predication** to conditionally execute insns in B and C
  - Schedule

Research: Frames

- New experimental scheduling construct: **frame**
  - rePLay [Patel+Lumetta]
  - Frame: an **atomic** superblock
    - Atomic means all or nothing, i.e., **transactional**
  - Two new insns
    - **`begin_frame`**: start buffering insn results
    - **`commit_frame`**: make frame results permanent
    - Hardware support required for buffering
  - Any branches out of frame: **abort the entire thing**
  + Eliminates nastiest part of trace scheduling … nasty repair code
    - If frame path is wrong just jump to original basic block code
    - Repair code still exists, but it's just the original code

Frames

Frame
```
8: begin_frame
0: ldf Y(r1),f2
2: ldf W(r1),f2
5: ldf X(r1),f4
6: mulf f4,f2,f6
1: fbne f2,0
7: stf f6,Z(r1)
9: commit_frame
```

Repair Code
```
0: ldf Y(r1),f2
1: fbne f2,4
2: ldf W(r1),f2
3: jump 5
4: stf f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```
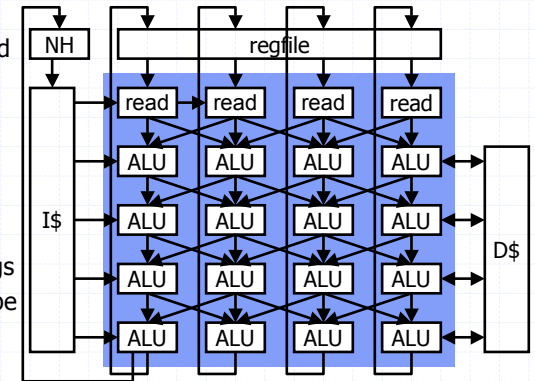
- What about frame optimizations?
  + Load-branch optimizations can be done without support
    - Natural branch "undo"
  - Load-store optimizations still require ISA support
    - Fixup code still simpler

Research: Grid Processor

- **Grid processor architecture** (aka TRIPS)
  - [Nagarajan, Sankaralingam, Burger+Keckler]
  - EDGE (Explicit Dataflow Graph Execution) execution model
  - Holistic attack on many fundamental superscalar problems
    - Specifically, the nastiest one: N^2 bypassing
    - But also N^2 dependence check
    - And wide-fetch + branch prediction
  - **Two-dimensional VLIW**
    - Horizontal dimension is insns in one parallel group
    - Vertical dimension is several vertical groups
  - Executes atomic hyperblocks

  - IBM looking into building it

CIS 501 (Martin/Roth): Multiple Issue and Static Scheduling

Grid Processor

- Components
  - next h-block logic/predictor (NH), I$, D$, regfile
  - NxN ALU grid: here 4x4
- Pipeline stages
  - Fetch h-block to grid
  - Read registers
  - Execute/memory
    - Cascade
  - Write registers
- Block atomic
  - No intermediate regs
  - Grid limits size/shape

Grid Processor SAXPY

```
read r2,0    read f1,0    read r1,0,1  nop
pass 0       pass 1       pass -1,1    ldf X,-1
pass 0       pass 0,1     mulf 1       ldf Y,0
pass 0       addi         pass 1       addf 0
blt          nop          pass 0,r1    stf Z
```
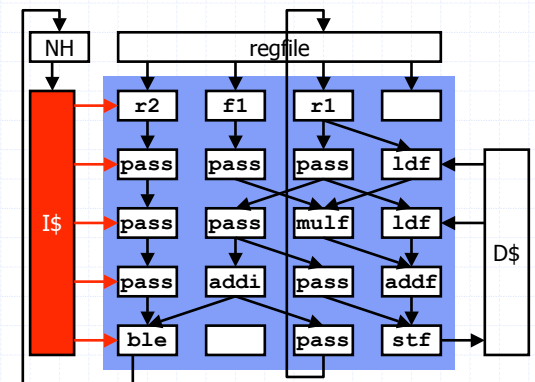
- An h-block for this Grid processor has 5 4-insn words
  - The unit is all 5
- Some notes about Grid ISA
  - **read**: read register from register file
  - **pass**: null operation
  - **-1,0,1**: routing directives send result to next word
    - one insn left (**-1**), insn straight down (**0**), one insn right (**1**)
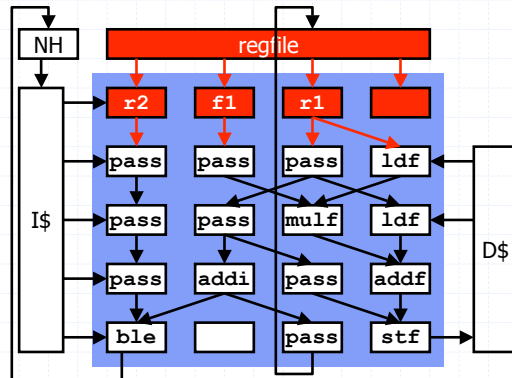   - Directives specify value flow, no need for interior registers

---

Grid Processor SAXPY Cycle 1

- Map hyperblock to grid

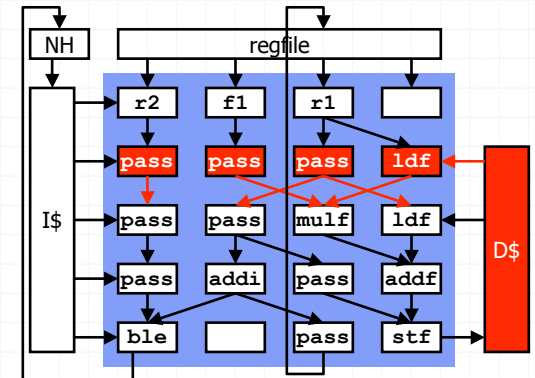## Grid Processor SAXPY Cycle 2

- Read registers

Grid Processor SAXPY Cycle 3

- Execute first grid row
- Execution proceeds in "data flow" fashion
  - Not lock step

Grid Processor SAXPY
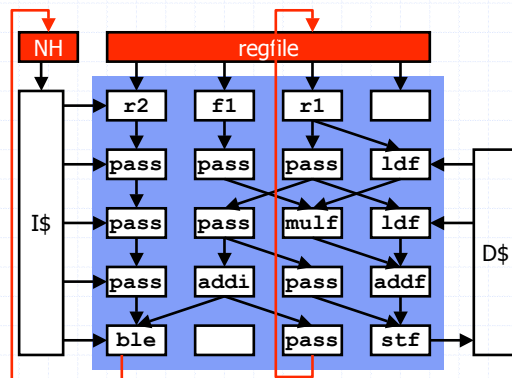
- When all instructions are done
  - Write registers and next hyperblock PC

Grid Processor SAXPY Performance

- Performance
  - 1 cycle fetch
  - 1 cycle read regs
  - 8 cycles execute
  - 1 cycle write regs
  - 11 cycles total (same)
- Utilization
  - 7 / (11 * 16) = 4%

- What's the point?
  + Simpler components
  + Faster clock?

Grid Processor Pros and Cons

+ Naturally aligned I$
+ No hardware dependence checks period
  - Insns explicitly encode rather than hardware reconstruct
  - Still get dynamic issue
+ **Simple, forward only, short-wire bypassing**
  - No wraparound routing, no metal layer crossings, low input muxes
− Code size
  - Lots of `nop` and `pass` operations
− Poor scheduling between hyperblocks
− Non-compatibility
  - Code assumes horizontal and vertical grid layout
  - Overcome with transparent dynamic translation? Like TransMeta
− Utilization
  - Overcome by multiple concurrent executing hyperblocks

Static Scheduling Summary

- Goal: increase scope to find more independent insns

- Loop unrolling
  + Simple
  − Expands code size, can't handle recurrences or non-loops

- Trace scheduling
  - Superblocks and hyperblocks
  + Works for non-loops
  − More complex, requires ISA support for speculation and predication
  − Requires nasty repair code

Multiple Issue Summary

- Problem spots
  - Wide fetch + branch prediction → trace cache?
  - N2 dependence cross-check
  - N2 bypass → clustering?

- Implementations
  - Statically scheduled superscalar
  - VLIW/EPIC
  - Research: Grid Processor

- What's next:
  - Finding more ILP by relaxing the in-order execution requirement