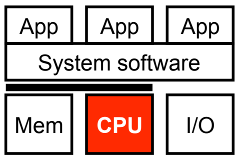
CIS 501
Computer Architecture

Unit 7: Superscalar

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: 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 cores

This Unit: Superscalar Execution

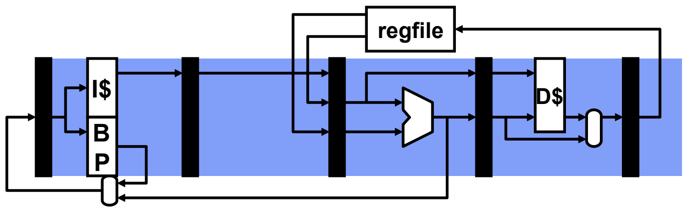


- Superscalar scaling issues
  - Multiple fetch and branch prediction
  - Dependence-checks & stall logic
  - Wide bypassing
  - Register file & cache bandwidth
- Multiple-issue designs
  - "Superscalar"
  - VLIW and EPIC (Itanium)
- Research: Grid Processor

Readings

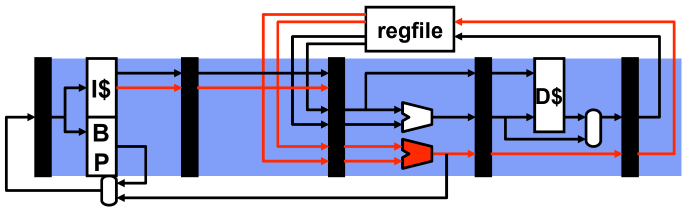
- H+P
  - TBD
- Paper
  - Edmondson et al., "Superscalar Instruction Execution in the 21164 Alpha Microprocessor"

Scalar Pipeline and the Flynn Bottleneck



- So far we have looked at **scalar pipelines**
  - One instruction per stage
    - With control speculation, bypassing, etc.
  - – 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 called **superscalar**
  - Two instructions per stage at once, or three, or four, or eight…
  - **"Instruction-Level Parallelism (ILP)"** [Fisher, IEEE TC'81]
- Today, typically "4-wide" (Intel Core 2, AMD Opteron)
  - Some more (Power5 is 5-issue; Itanium is 6-issue)
  - Some less (dual-issue is common for simple cores)

Superscalar Pipeline Diagrams - Ideal

scalar	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)→r2`	F	D	X	M	W							
`lw 4(r1)→r3`		F	D	X	M	W						
`lw 8(r1)→r4`			F	D	X	M	W					
`add r14,r15→r6`				F	D	X	M	W				
`add r12,r13→r7`					F	D	X	M	W			
`add r17,r16→r8`						F	D	X	M	W		
`lw 0(r18)→r9`							F	D	X	M	W	

2-way superscalar	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)→r2`	F	D	X	M	W							
`lw 4(r1)→r3`	F	D	X	M	W							
`lw 8(r1)→r4`		F	D	X	M	W						
`add r14,r15→r6`		F	D	X	M	W						
`add r12,r13→r7`			F	D	X	M	W					
`add r17,r16→r8`			F	D	X	M	W					
`lw 0(r18)→r9`				F	D	X	M	W				

Superscalar Pipeline Diagrams - Realistic

scalar	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)→r2`	F	D	X	M	W							
`lw 4(r1)→r3`		F	D	X	M	W						
`lw 8(r1)→r4`			F	D	X	M	W					
`add r4,r5→r6`				F	d*	D	X	M	W			
`add r2,r3→r7`						F	D	X	M	W		
`add r7,r6→r8`							F	D	X	M	W	
`lw 0(r8)→r9`								F	D	X	M	W

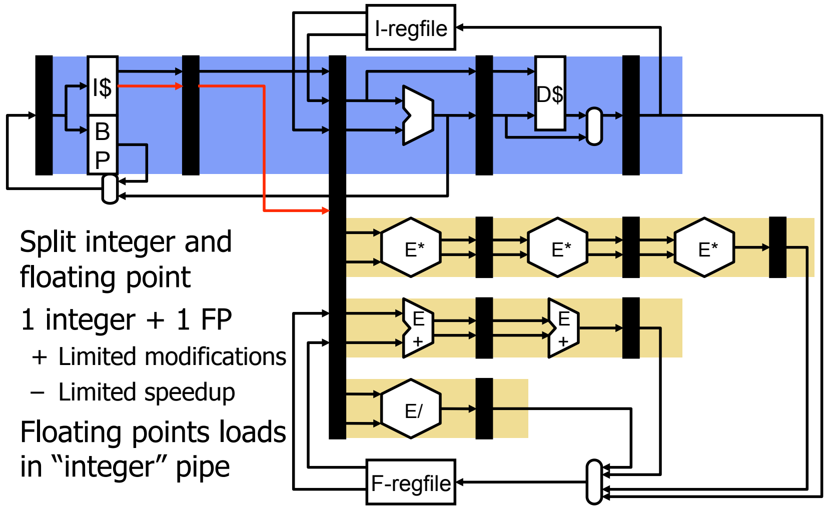
2-way superscalar	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)→r2`	F	D	X	M	W							
`lw 4(r1)→r3`	F	D	X	M	W							
`lw 8(r1)→r4`		F	D	X	M	W						
`add r4,r5→r6`		F	d*	d*	D	X	M	W				
`add r2,r3→r7`			F	d*	D	X	M	W				
`add r7,r6→r8`					F	D	X	M	W			
`lw 0(r8)→r9`					F	d*	D	X	M	W		

Superscalar CPI Calculations

- Base CPI for scalar pipeline is 1
- **Base CPI for N-way superscalar pipeline is 1/N**
  - – Amplifies stall penalties
  - Assumes no data stalls (an overly optmistic assumption)

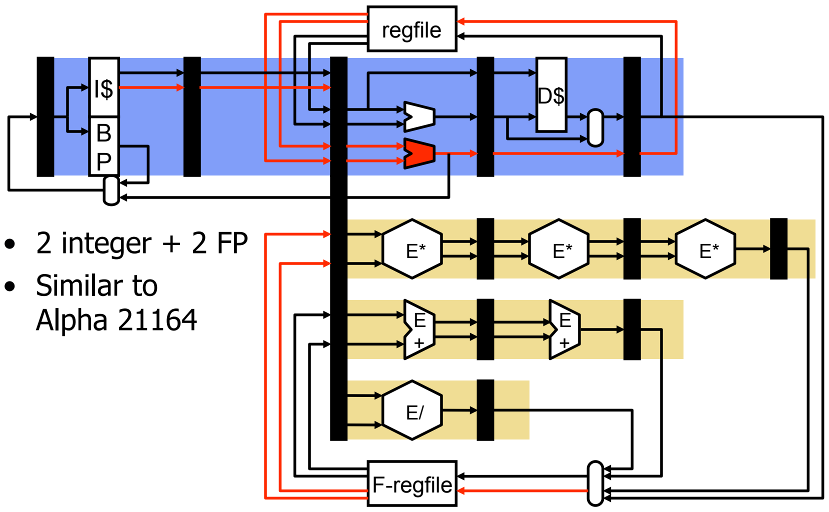
- Example: Branch penalty calculation
  - 20% branches, 75% taken, no explicit branch prediction
- Scalar pipeline
  - 1 + 0.2*0.75*2 = 1.3 → 1.3/1 = 1.3 → 30% slowdown
- 2-way superscalar pipeline
  - **0.5** + 0.2*0.75*2 = 0.8 → 0.8/0.5 = 1.6 → 60% slowdown
- 4-way superscalar
  - **0.25** + 0.2*0.75*2 = 0.55 → 0.55/0.25 = 2.2 → 120% slowdown

Simplest Superscalar: Split Floating Point



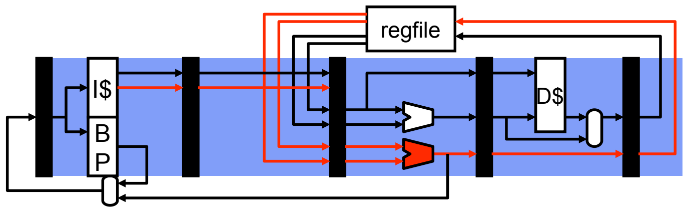
- Split integer and floating point
- 1 integer + 1 FP
  - + Limited modifications
  - – Limited speedup
- Floating points loads in "integer" pipe

A Four-issue Pipeline (2 integer, 2 FP)



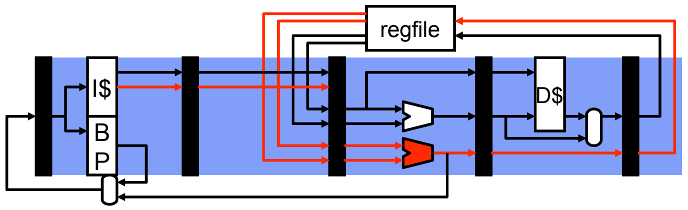
- 2 integer + 2 FP
- Similar to Alpha 21164

A Typical 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)

A Typical 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
  - Single load per cycle (stall at decode) probably okay for dual issue
  - Alternative: add a read port to data cache
    - Larger area, latency, power, cost, complexity

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
    - Each port needs its own set of address and data wires
  - 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

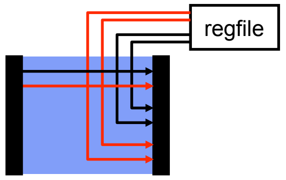
How Much ILP is There?

- The compiler tries to "schedule" code to avoid stalls
  - Even for scalar machines (to fill load-use delay slot)
  - Even harder to schedule multiple-issue (superscalar)

- How much ILP is common?
  - Greatly depends on the application
    - Consider memory copy
    - Unroll loop, lots of independent operations
  - Other programs, less so

- Even given unbounded ILP, superscalar has limits
  - IPC (or CPI) vs clock frequency trade-off
  - Given these challenges, what is reasonable N? 3 or 4 today

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 ∝ #ports)
  - + Actually less than 2N, most values come from bypasses
  - More about this in a bit
- What about the **stall logic**?

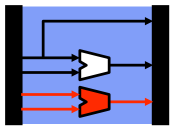
N^2 Dependence Cross-Check

- 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": ∝ 2N
- Now: same logic for a 2-wide pipeline

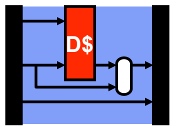
  X/M_1.op==LOAD && (D/X_1.rs1==X/M_1.rd || D/X_1.rs2==X/M_1.rd) ||
  X/M_1.op==LOAD && (D/X_2.rs1==X/M_1.rd || D/X_2.rs2==X/M_1.rd) ||
  X/M_2.op==LOAD && (D/X_1.rs1==X/M_2.rd || D/X_1.rs2==X/M_2.rd) ||
  X/M_2.op==LOAD && (D/X_2.rs1==X/M_2.rd || D/X_2.rs2==X/M_2.rd)
  - Eight "terms": ∝ $2N^2$
    - **N^2 dependence cross-check**
  - Not quite done, also need
    - D/X_2.rs1==D/X_1.rd || D/X_2.rs2==D/X_1.rd

Wide Execute



- What is involved in executing 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
    - Alpha 21164: 2 integer (including 2 loads) + 2 FP

Wide Memory Access



- What about multiple loads/stores per cycle?
  - Probably only necessary on processors 4-wide or wider
  - More important to support multiple loads than multiple stores
    - Insn mix: loads (~20–25%), stores (~10–15%)
  - Alpha 21164: two loads *or* one store per cycle

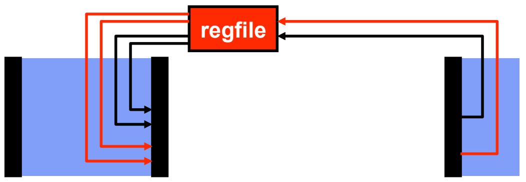
D$ Bandwidth: Multi-Porting, Replication

- How to provide additional D$ bandwidth?
  - Have already seen split I$/D$, but that gives you just one D$ port
  - How to provide a second (maybe even a third) D$ port?
- Option#1: **multi-porting**
  - + Most general solution, any two accesses per cycle
  - – Expensive in terms of latency, area (cost), and power
- Option #2: **replication**
  - Additional read bandwidth only, but writes must go to all replicas
  - + General solution for loads, no latency penalty
  - – Not a solution for stores (that's OK), area (cost), power penalty
    - Is this what Alpha 21164 does?

D$ Bandwidth: Banking

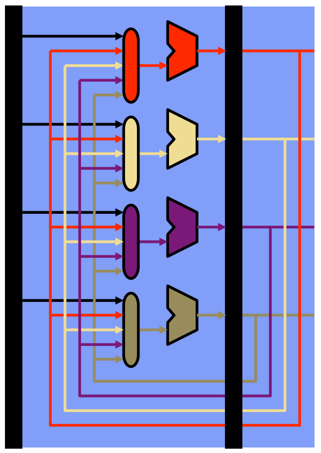
- Option#3: **banking** (or **interleaving**)
  - Divide D$ into "banks" (by address), 1 access/bank-cycle
  - **Bank conflict**: two accesses to same bank → one stalls
  - + No latency, area, power overheads (latency may even be lower)
  - + One access per bank per cycle, **assuming no conflicts**
  - – Complex stall logic → address not known until execute stage
  - – To support N accesses, need 2N+ banks to avoid frequent conflicts
- Which address bit(s) determine bank?
  - Offset bits? Individual cache lines spread among different banks
    - + Fewer conflicts
    - – Must replicate tags across banks, complex miss handling
  - Index bits? Banks contain complete cache lines
    - – More conflicts
    - + Tags not replicated, simpler miss handling

Wide Register Read/Write



- How many register file ports to execute N insns per cycle?
  - Nominally, 2N read + N write (2 read + 1 write per insn)
    - – Latency, area ∝ #ports^2
  - In reality, fewer than that
    - Read ports: many values come from bypass network
    - Write ports: stores, branches (35% insns) don't write registers
- Replication works great for regfiles (used in Alpha 21164)
- Banking? Not so much

Wide Bypass

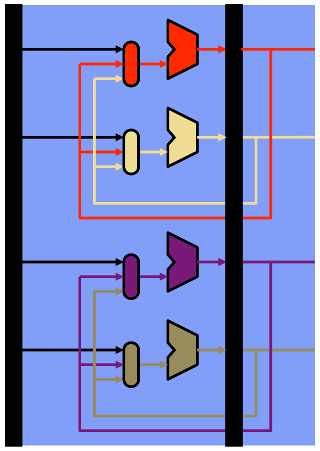


- **N^2 bypass network**
  - – N+1 input muxes at each ALU input
  - – N^2 point-to-point connections
  - – Routing lengthens wires
  - – Expensive metal layer crossings
  - – Heavy capacitive load
  - And this is just one bypass stage (MX)!
    - There is also WX bypassing
    - Even more for deeper pipelines
  - One of the big problems of superscalar
- Implemented as bit-slicing
  - 64 1-bit bypass networks
  - Mitigates routing problem somewhat

Aside: Not All N^2 Created Equal

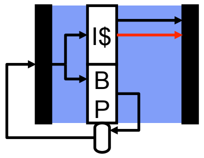
- N^2 bypass vs. N^2 dependence cross-check
  - Which is the bigger problem?
- N^2 bypass … by far
  - 32- or 64- bit quantities (vs. 5-bit)
  - Multiple levels (MX, WX) of bypass (vs. 1 level of stall logic)
  - Must fit in one clock period with ALU (vs. not)
- Dependence cross-check not even 2nd biggest N^2 problem
  - Regfile is also an N^2 problem (think latency where N is #ports)
  - And also more serious than cross-check

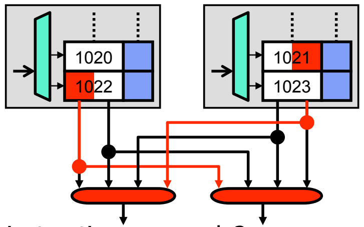
Clustering



- **Clustering**: mitigates N^2 bypass
  - Group ALUs into **K** clusters
  - Full bypassing within a cluster
  - Limited bypassing between clusters
    - With 1 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
  - Each cluster has register file replica

Wide Fetch - Sequential Instructions



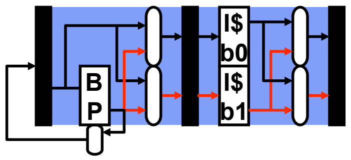


- What is involved in fetching multiple instructions per cycle?
- In same cache block? → no problem
  - Favors larger block size (independent of hit rate)
- Compilers align basic blocks to I$ lines (`.align` assembly directive)
  – Reduces I$ capacity
  + Increases fetch bandwidth utilization (more important)
- 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 Non-Sequential Fetch

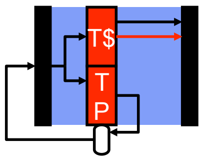
- Two related questions
  - How many branches predicted per cycle?
  - Can we fetch across the branch if it is predicted "taken"?
- Simplest, most common organization: "1" and "No"
  - One prediction, discard post-branch insns if prediction is "taken"
  – Lowers effective fetch width and IPC
  - Average 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
  - Reduce taken branch frequency (e.g., unroll loops)

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, next PC and next-next 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

Pentium4 Trace Cache

- Pentium4 has a trace cache …
  - But doesn't use it to solve the branch prediction/fetch problem
  - Uses it to solve decode problem instead
  - Traces contain decoded RISC μops (not CISC x86 insns)
  - Traces are short (3 μops each)
- What is the "decoding" problem?
  - Breaking x86 insns into μops is slow and area-/energy-consuming
  - Especially problematic is converting x86 insns into mulitple μops
    - Average μop/x86 insns ratio is 1.6–1.7
  - Pentium II (and III) only had 1 multiple-μop decoder
    - Big performance hit vis-à-vis AMD's Athlon (which had 3)
  - Pentium4 uses T$ to "simulate" multiple multliple-μop decoders
  - And to shorten pipeline (which is still 22 stages)

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

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've 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 You?

+ Simpler I$/branch prediction

+ Slightly simpler dependence check logic

- Doesn't help bypasses or regfile
  - Which are the much bigger problems
  - Although clustering and replication can help VLIW, too

– Not compatible across machines of different widths
  - Is non-compatibility worth all of this?

- PS did 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)
  - Still does not address bypassing or register file issues

Trends in Single-Processor Multiple Issue

	486	Pentium	PentiumII	Pentium4	Itanium	ItaniumII	Core2
Year	1989	1993	1998	2001	2002	2004	2006
Width	1	2	3	3	3	6	4

- Issue width has saturated at 4-6 for high-performance cores
  - Canceled Alpha 21464 was 8-way issue
  - No justification for going wider
  - Out-of-order execution (or EPIC) needed to exploit 4-6 effectively
- For high-performance/power cores, issue width is 1-2
  - Out-of-order execution not needed
  - Multi-threading (a little later) helps cope with cache misses

Multiple Issue Redux

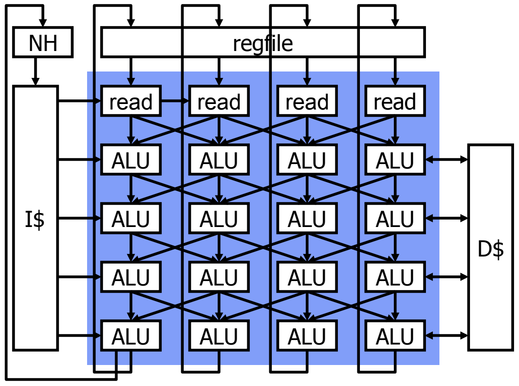
- Multiple issue
  - Needed to expose insn level parallelism (ILP) beyond pipelining
  - Improves performance, but reduces utilization
  - 4-6 way issue is about the peak issue width currently justifiable
- Problem spots
  - Fetch + branch prediction → trace cache?
  - N^2 bypass → clustering?
  - Register file → replication?
- Implementations
  - (Statically-scheduled) superscalar, VLIW/EPIC
- Are there more radical ways to address these challenges?

Research: Grid Processor

- **Grid processor** (TRIPS) [Nagarajan+, MICRO'01]
  - 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 code blocks
    - Uses predication and special scheduling to avoid taken branches
  - UT-Austin research project
  - Fabricated an actual chip with help from IBM

Grid Processor

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



Aside: 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
```

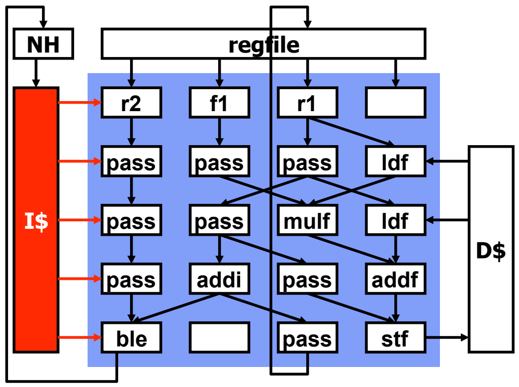
Grid Processor SAXPY

```
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
```

- A code block for this Grid processor has 5 4-insn words
  - Atomic unit of execution
- 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

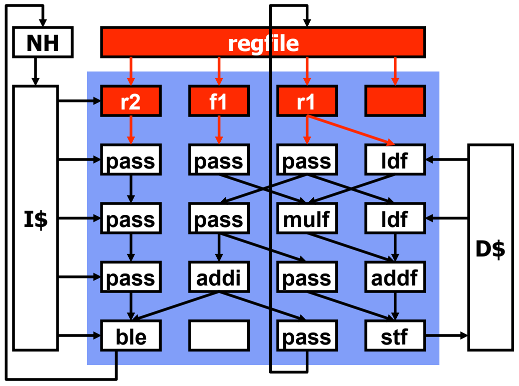
Grid Processor SAXPY Cycle 1

- Map code block to grid



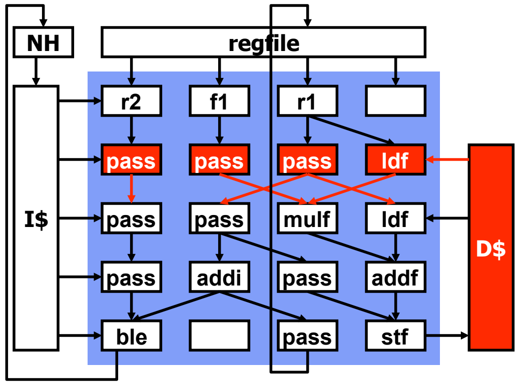
Grid Processor SAXPY Cycle 2

- Read registers



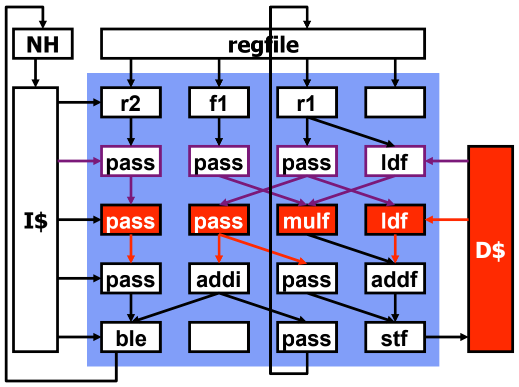
Grid Processor SAXPY Cycle 3

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



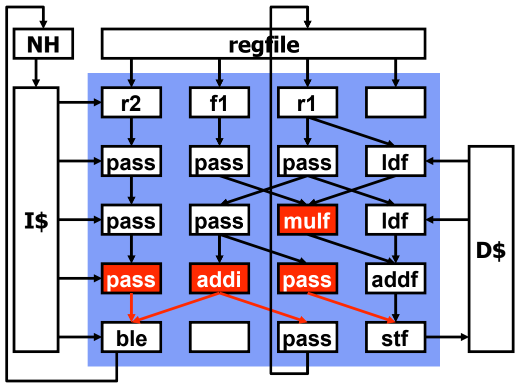
Grid Processor SAXPY Cycle 4

- Execute second grid row



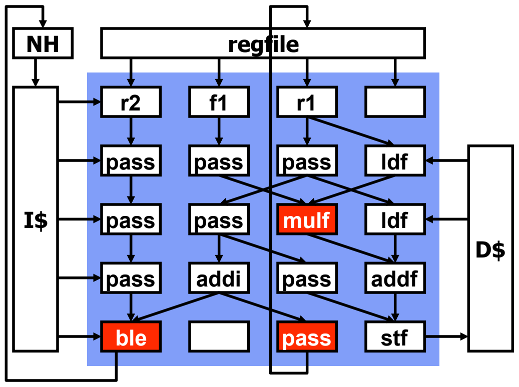
Grid Processor SAXPY Cycle 5

- Execute third grid row
  - Recall, `mulf` takes 5 cycles



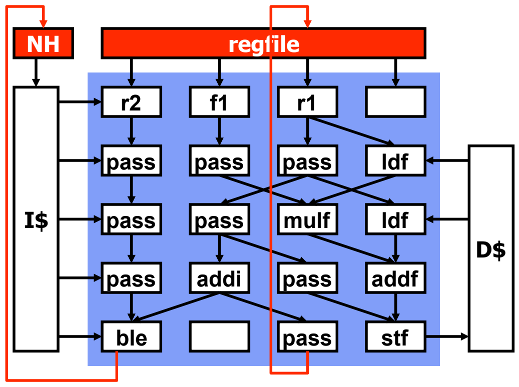
Grid Processor SAXPY Cycle 6

- Execute third grid row



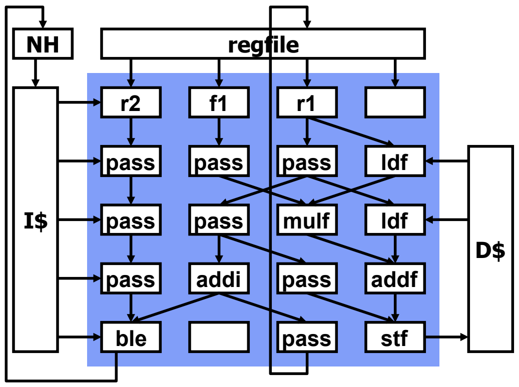
Grid Processor SAXPY

- When all instructions are done
  - Write registers and next code block PC



Grid Processor SAXPY Performance

- Performance
  - 1 cycle fetch
  - 1 cycle read regs
  - 8 cycles execute
  - 1 cycle write regs
  - 11 cycles total
- Utilization
  - 7 / (11 * 16) = 4%
- What's the point?
  - + Simpler components
  - + Faster clock?



Grid Processor Redux

+ No hardware dependence checks … period
- Insn placement encodes dependences, 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

– Non-compatibility
- Code assumes horizontal *and vertical* grid layout

– No scheduling between hyperblocks
- Can be overcome, but is pretty nasty

– Poor utilization
- Overcome by multiple concurrent executing hyperblocks

- Interesting: TRIPS has morphed into something else
  - Grid is gone, replaced by simpler "window scheduler"
  - Forward nature of ISA exploited to create tiled implementations

Next Up…

- Extracting more ILP via…
  - Static scheduling by compiler
  - Dynamic scheduling in hardware