Instruction Level Parallelism II: Superscalar Execution

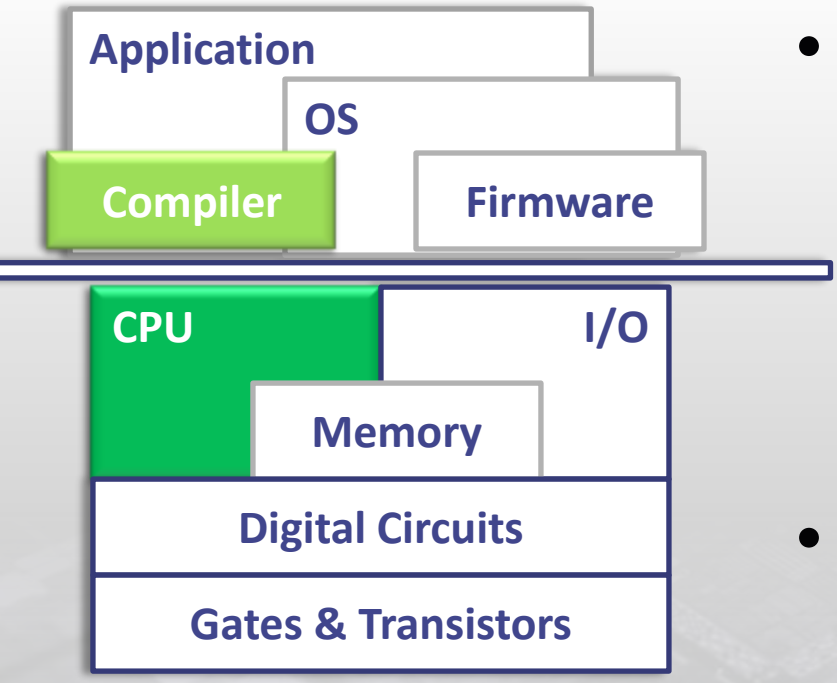
Readings:

H&P: Chapter 2.3, 2.7-2.12

H&P 3rd Edition: Chapter 4

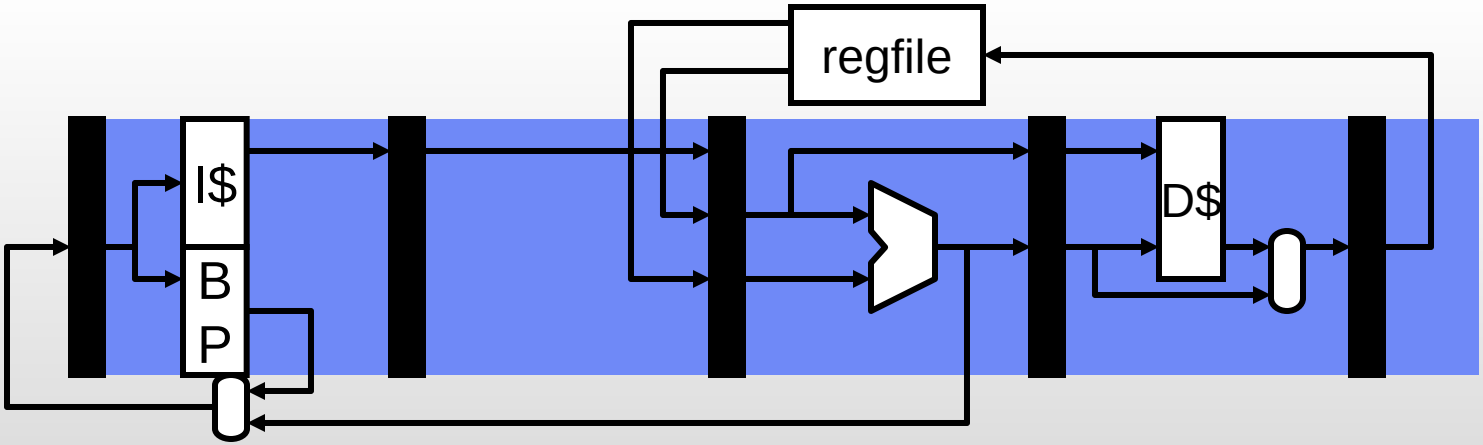
Paper: Edmondson et al., “Superscalar Instruction Execution in the 21164 Alpha Microprocessor” (<Memory Instructions)

This Unit: Superscalar Execution



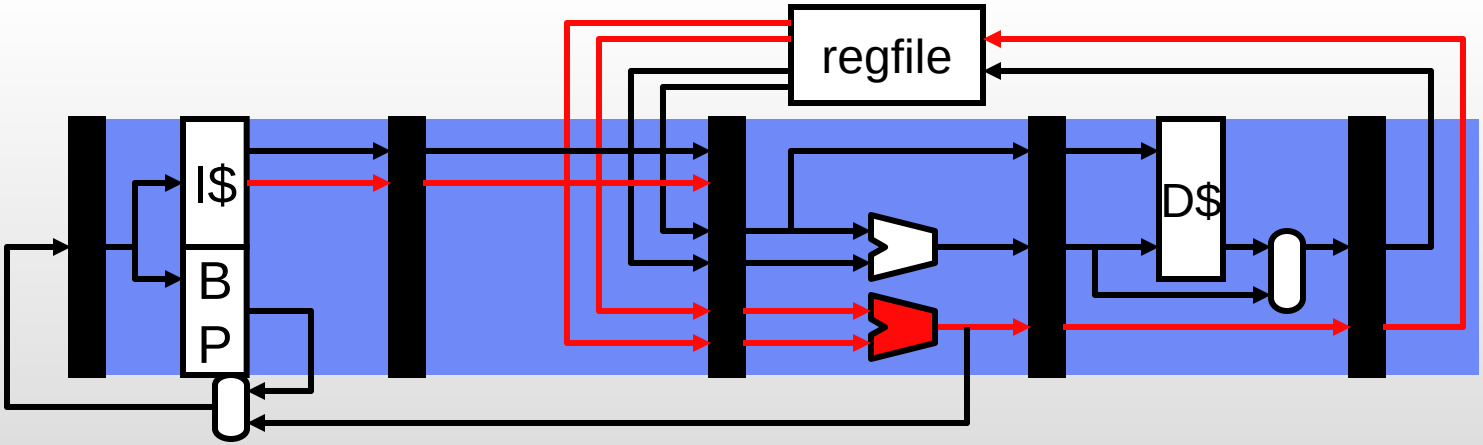
- Superscalar scaling issues
  - Multiple fetch and branch prediction
  - Dependence-checks & stall logic
  - Wide bypassing
  - Register file & cache bandwidth
- Multiple-issue designs
  - "Superscalar"
  - VLIW and EPIC (Itanium)
- Static scheduling

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 such as Atom, ARM A8)

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

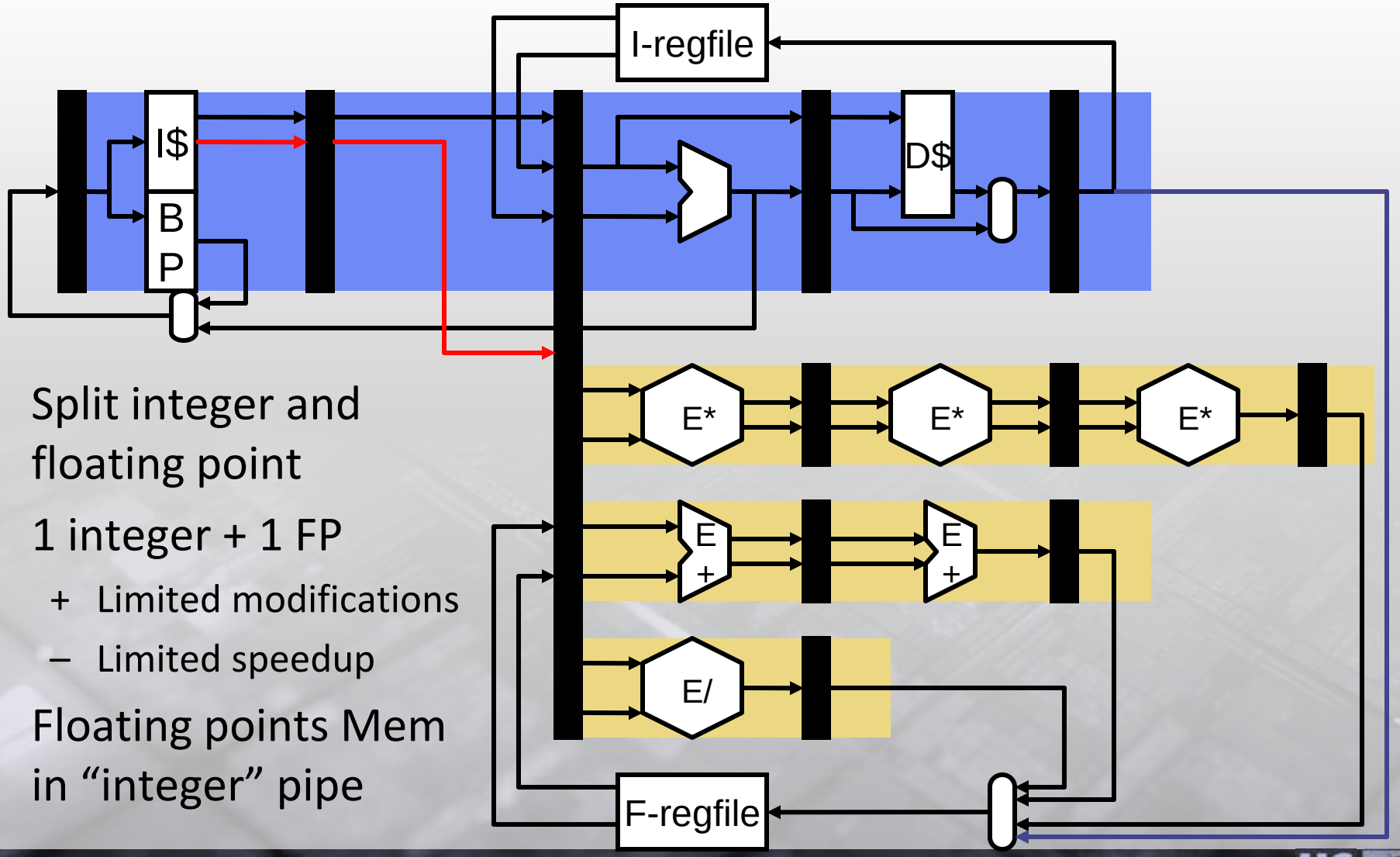
	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)➔r2`	F	D	X	M	W							
`lw 4(r1)➔r3`	F	D	X	M	W							
`lw 8(r1)➔r4`		F	D	X	M	W						
`add r4,r5➔r6`		F	d*	d*	D	X	M	W				
`add r2,r3➔r7`			F	D	X	M	W					
`add r7,r6➔r8`					F	D	X	M	W			
`lw 0(r8)➔r9`					F	d*	D	X	M	W		

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

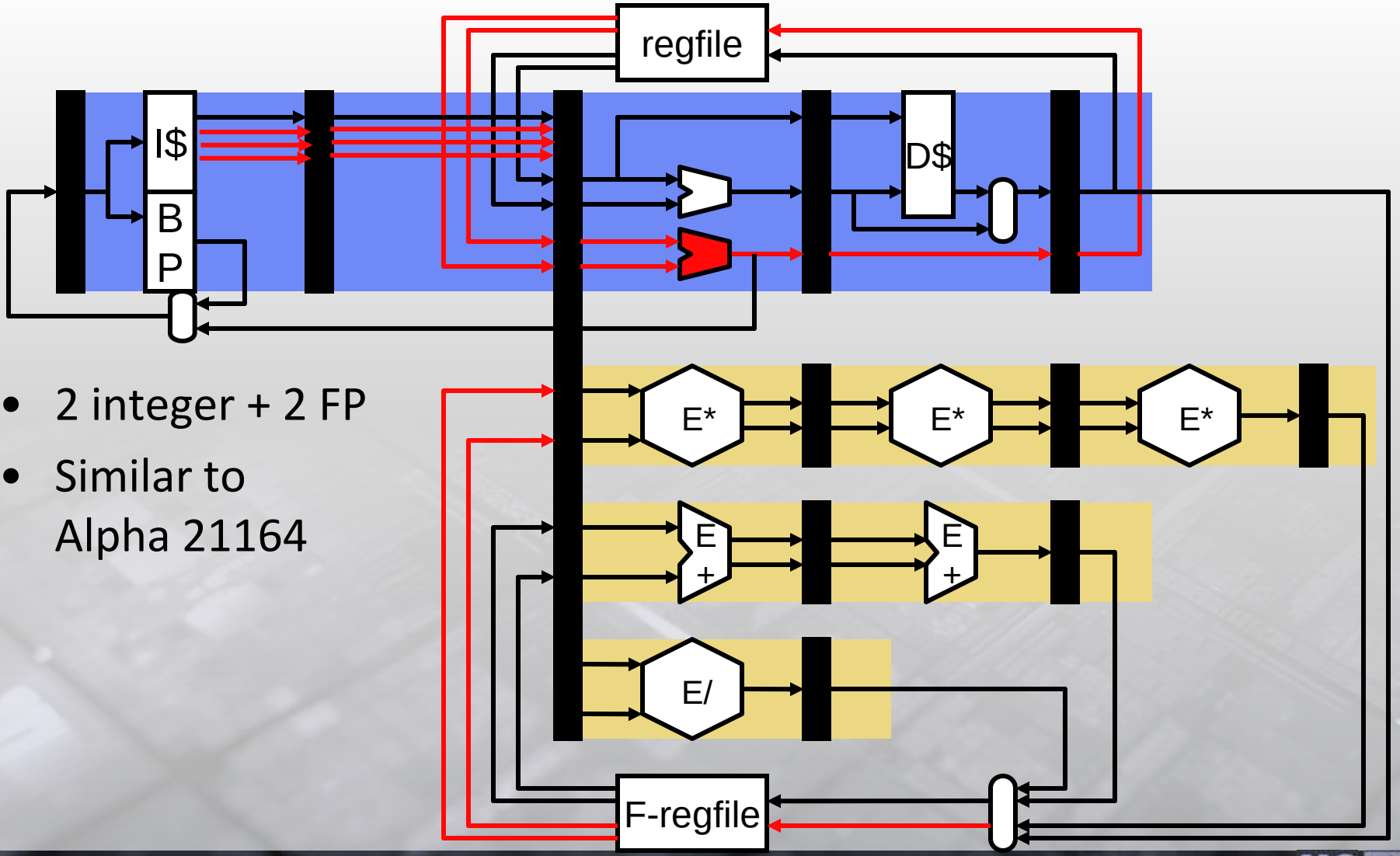
Simplest Superscalar: Split Floating Point



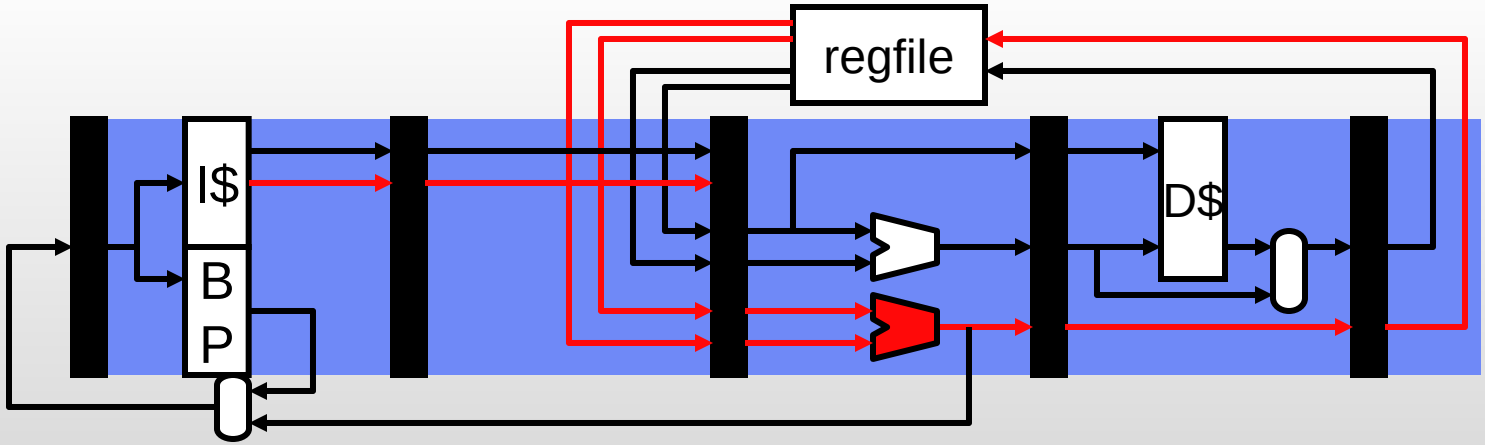
- Split integer and floating point
- 1 integer + 1 FP
  - \+ Limited modifications
  - – Limited speedup
- Floating points Mem in "integer" pipe

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

- 2 integer + 2 FP
- Similar to Alpha 21164

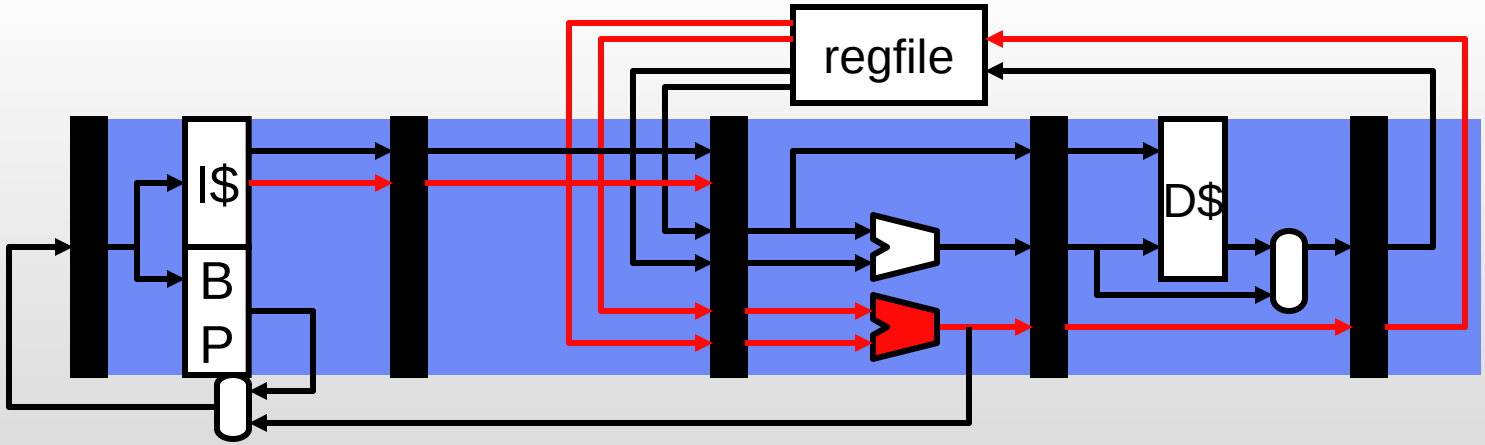


A Typical Dual-Issue Pipeline



- 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 (how many stall cycles with full bypassing?)
  - 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 ALUs are easy, but bypass paths are expensive
- Memory unit
  - Single load per cycle (stall at decode) probably okay for dual issue
  - Alternative: add a additional 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 - ~$O(N^2)$ 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
  - ~O(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

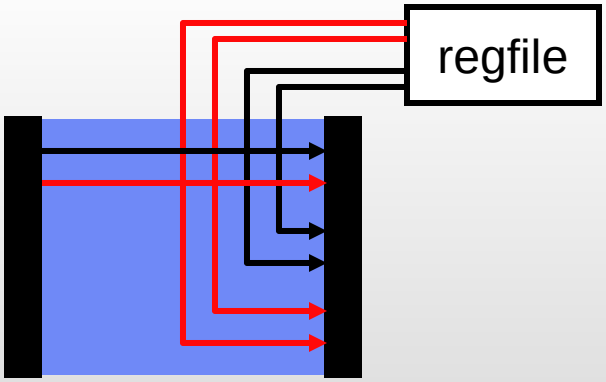
How Much ILP is There?

- The compiler tries to “schedule” code to avoid stalls
  - Even for scalar machines (Example?)
  - 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

FE: 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
  - More about this in a bit
- What about the **stall logic**? (e.g. RAW on load)

N^2 Dependence Cross-Check (e.g)

- Stall logic for 1-wide pipeline with full bypassing
  - Full bypassing → load/use stalls only (Assuming stall logic at Decode, and W→D natural bypass through register file)

    D/X.op==LOAD && (F/D.rs1==X/M.rd || F/D.rs2==D/X.rd)
  - Two "terms": $\propto 2N$
- Now: same logic for a 2-wide pipeline

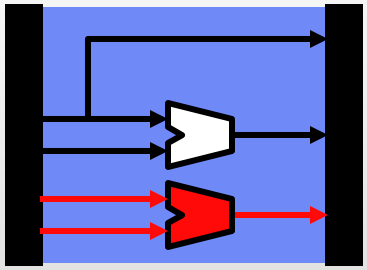
  D/X_1.op==LOAD && (F/D_1.rs1==D/X_1.rd || F/D_1.rs2==D/X_1.rd) ||

  D/X_1.op==LOAD && (F/D_2.rs1==D/X_1.rd || F/D_2.rs2==D/X_1.rd) ||

  D/X_2.op==LOAD && (F/D_1.rs1==D/X_2.rd || F/D_1.rs2==D/X_2.rd) ||

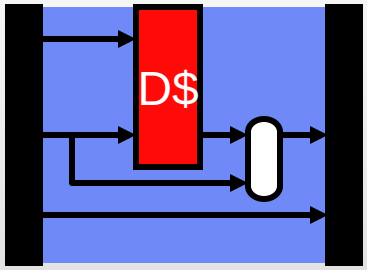
  D/X_2.op==LOAD && (F/D_2.rs1==D/X_2.rd || F/D_2.rs2==D/X_2.rd)
  - Eight "terms": $\propto 2N^2$
    - **N^2 dependence cross-check**
  - Not quite done, also need
    - F/D_2.rs1==F/D_1.rd || F/D_2.rs2==F/D_1.rd || F/D_1.rs1==F/D_2.rd || F/D_1.rs2==F/D_2.rd

FE: 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
      - What if 3 integer operations in the “bundle”?

FE: 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
  - – To increase bit-lines is expensive in terms of latency, area (cost), and power

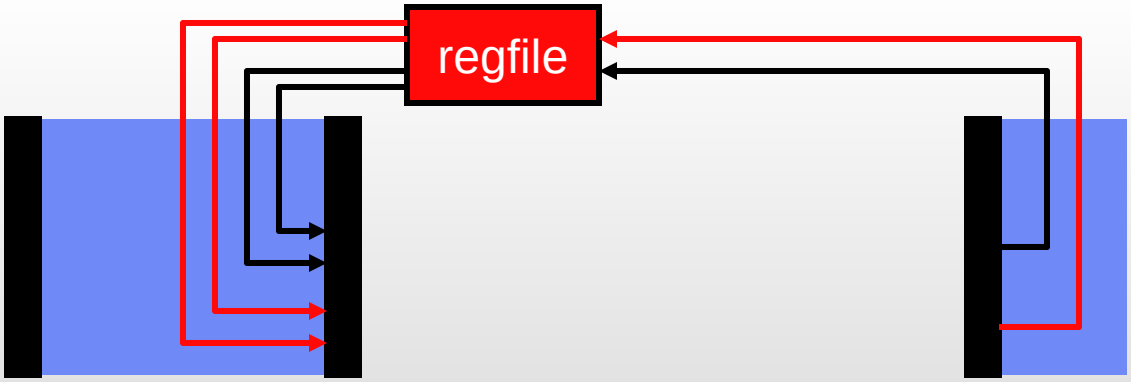
- Option #2: **replication**
  - Additional read bandwidth only, but writes must go to all replicas
  - \+ General solution for loads, no latency penalty
  - – Not a solution for stores (that's OK), area (cost), power penalty
    - Is this what Alpha 21164 does?

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

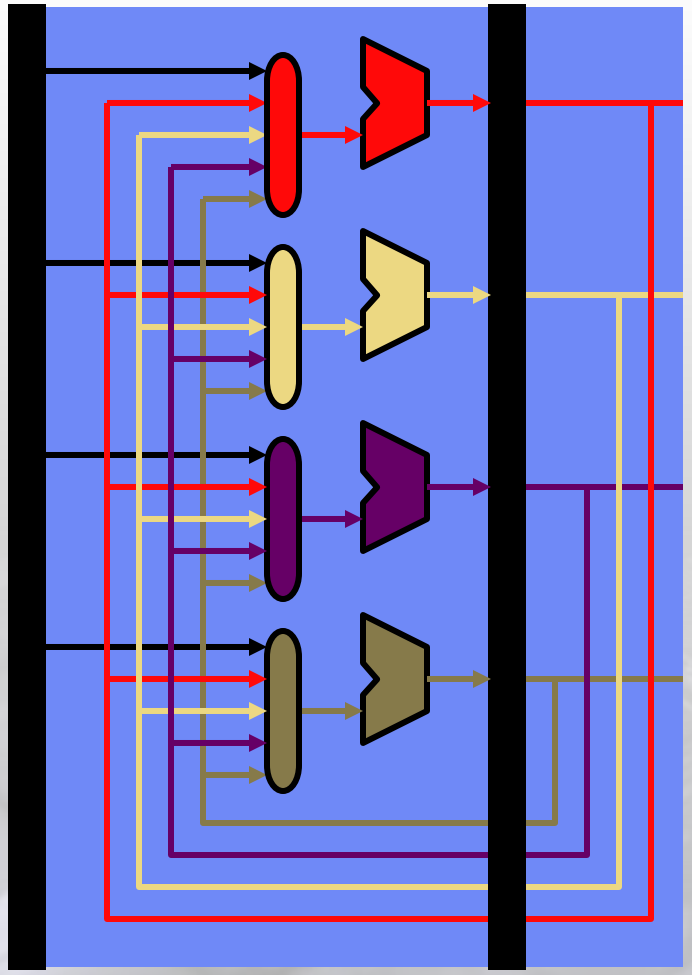
- Which address bit(s) determine bank?
  - (By column) Offset bits? Individual cache lines spread among different banks
    - \+ Fewer conflicts
    - – Must replicate tags across banks, complex miss handling
  - (By Row) Index bits? Banks contain complete cache lines
    - – More conflicts
    - \+ Tags not replicated, simpler miss handling

BE: 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 $\propto$ #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

BE: Wide Bypass

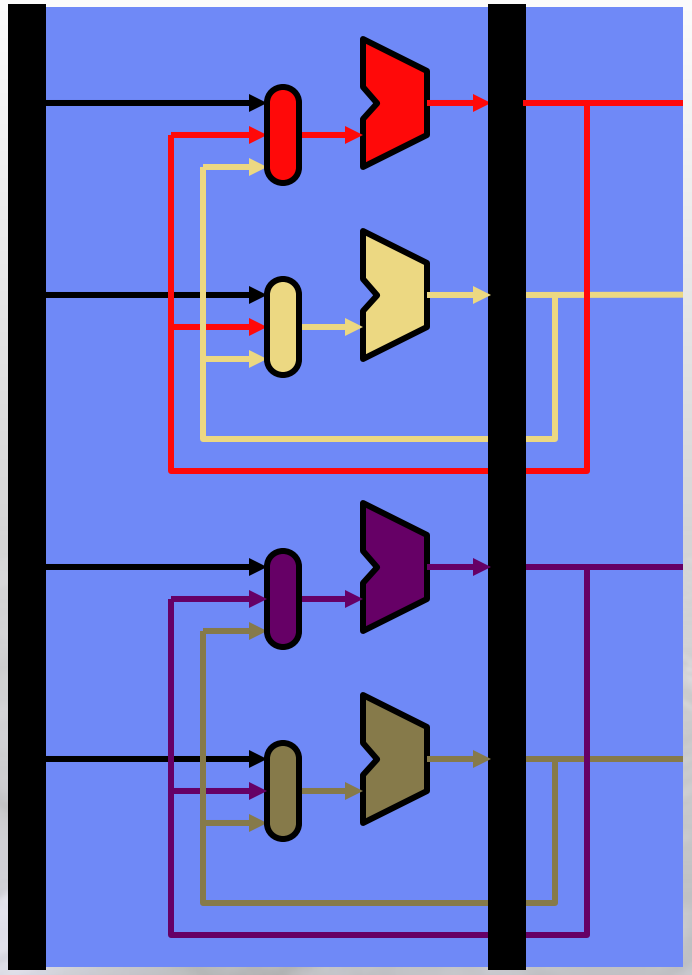


- **N^2 bypass network**
  - N+1 input muxes at each ALU input
  - N^2 point-to-point connections
  - Routing lengthens wires
  - Expensive metal layer crossings
  - **Heavy capacitive load**
  - And this is just one bypass stage (MX)!
    - There is also WX bypassing
    - Even more for deeper pipelines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

Aside: Not All N^2 Created Equal

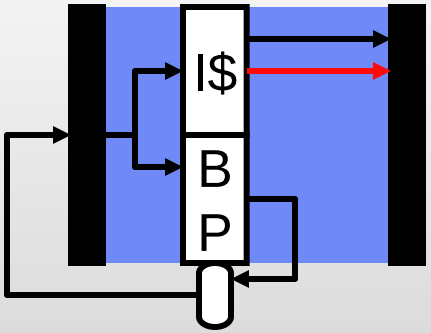
- N^2 bypass vs. N^2 dependence cross-check
  - Which is the bigger problem?
- N^2 bypass … by far
  - 32- or 64- bit quantities (vs. 5-bit)
  - Multiple levels (MX, WX) of bypass (vs. 1 level of stall logic)
  - Must fit in one clock period with ALU (vs. not)
- Dependence cross-check not even 2nd biggest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

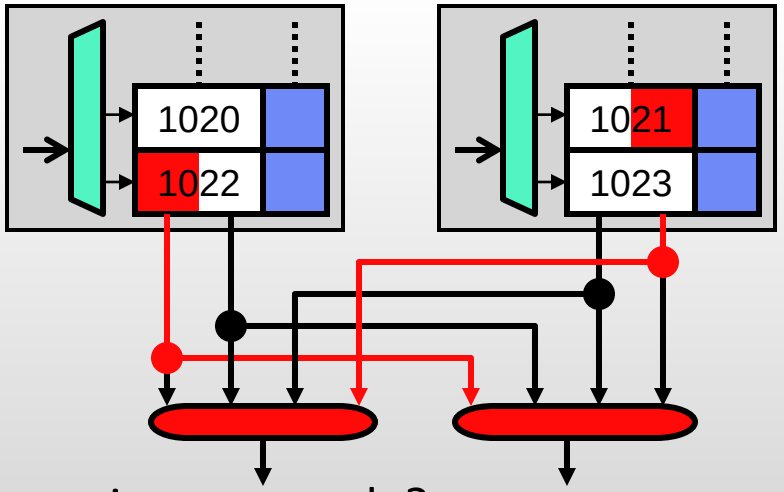
Clustering



- **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

FE/BE: 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 (Why?)
  - + 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)

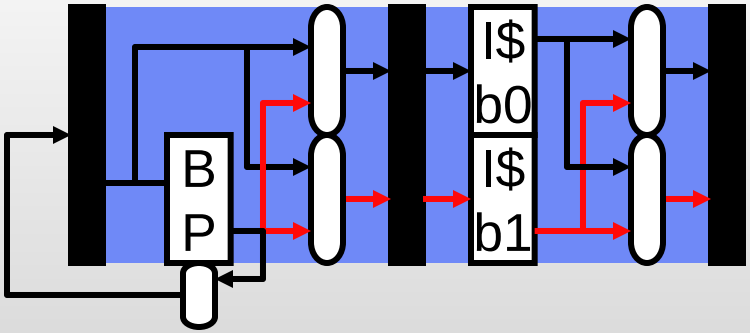
FE/BE: Wide Non-Sequential Fetch

- Two related questions
  - How many branches predicted per cycle?
  - Can we fetch across the branch (in the same cycle) 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, on average, ILP is limited to 5 (not 8)

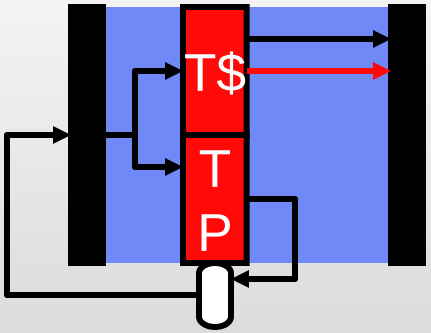
- Compiler can help
  - Reduce taken branch frequency (e.g., unroll loops)

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

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

	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

```
0: addi r1,4,r1
1: beq r1,#4
4: st r1,4(sp)
5: 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

FE: 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 Designs

Multiple-Issue Designs

- **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 DSP 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 DSP centric 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 **VL**IW meaning (**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
  - Memory wall makes no reasonable wider (We will see that in lab)
  - 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 $\rightarrow$ trace cache?
  - N^2 bypass $\rightarrow$ clustering?
  - Register file $\rightarrow$ replication?
- Implementations
  - (Statically-scheduled) superscalar, VLIW/EPIC
- Are there more radical ways to address these challenges?
  - See TRIPS processor at the appendix

SCHEDULING

ILP and Static Scheduling

- No point to having an N-wide pipeline…
- …if average number of parallel insns per cycle (ILP) << N
  - Performance is important…
  - … but so is **utilization**: actual performance / peak performance
  - Unutilized hardware still consumes power (begins to be itchy)
  - Unutilized hardware still consumes area
  - Unutilized hardware may slow down clock (clock vs. IPC) …
  - … or lengthen pipeline (IPC vs. IPC)
- Rest of unit: how compiler can help extract parallelism
  - Important for superscalar
  - Critical for VLIW/EPIC
- Next unit: how hardware can extract parallelism

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	X	M	W														
`addf f2,f3→f4`		F	p*	p*	D	d*	d*	d*	d*	E+	E+	W								
`stf f4→Z(r1)`			F	D	p*	p*	p*	p*	p*	d*	X	M	W							
`addi r1,4→r1`					F	p*	p*	p*	p*	D	X	M	W							
`blt r1,r2,0`					F	p*	p*	p*	p*	p*	D	X	M	W						
`ldf X(r1)→f1`										F	D	X	M	W						

- Dual issue pipeline
  - Same + any two insns per cycle + embedded taken branches
  - \+ **Performance**: 7 insns / 9 cycles = 0.78 IPC
  - – **Utilization**: 0.70 actual IPC / 2 peak IPC = 38%
  - – 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 adds 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: constants → constants + 4

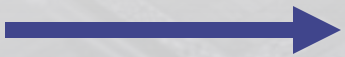
```
ldf X(r1)→f1
mulf f0,f1→f2
ldf Y(r1)→f3
addf f2,f3→f4
stf f4→Z(r1)
addi r1,4→r1
blt r1,r2,0
ldf X(r1)→f1
mulf f0,f1→f2
ldf Y(r1)→f3
addf f2,f3→f4
stf f4→Z(r1)
addi r1,4→r1
blt r1,r2,0
```

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

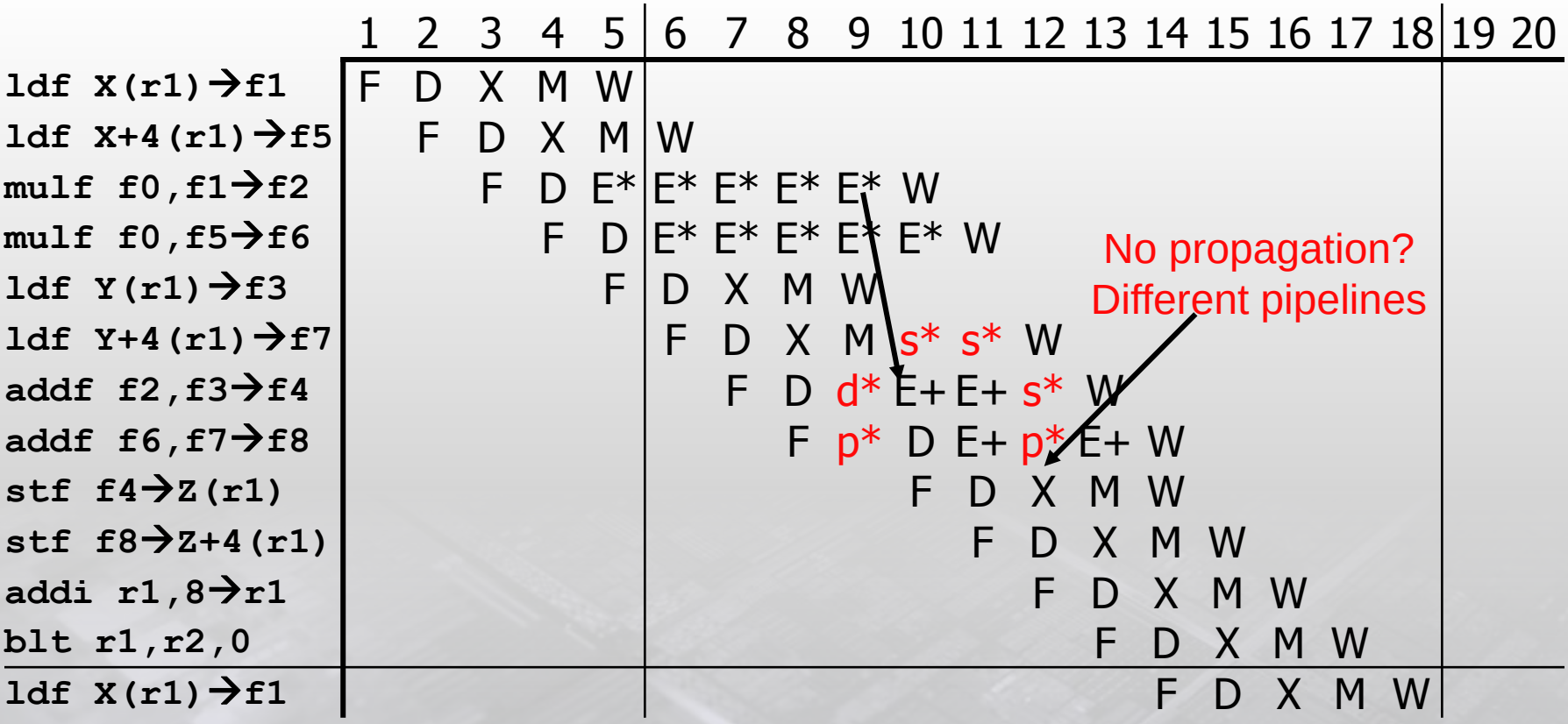
Unrolling SAXPY III: Rename Registers

- Pipeline scheduling causes WAR violations
  - Rename registers to correct

```
 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 r1,8→r1
 blt r1,r2,0
```

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

Unrolled SAXPY Performance/Utilization



+ 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

Dual-Issue Performance/Utilization

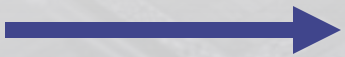
+ Performance: 12 insn / 10 cycles = 1.2 IPC
+ Utilization: 1.2 actual IPC / 2 peak IPC = 60%
+ **Speedup**: (2 * 9 cycles) / 10 cycles = 1.8

Loop Unrolling Shortcomings

- Static code growth → more I$ misses (limits degree of unrolling)
- Poor scheduling along "seams" of unrolled copies
- Need more registers to resolve WAR hazards
- **Doesn't handle recurrences** (inter-iteration dependences)

```
for (i=0;i<N;i++)
   X[i]=A*X[i-1];
```

```
ldf X-4(r1)→f1
mulf f0,f1→f2
stf f2→X(r1)
addi r1,4→r1
blt r1,r2,0
ldf X-4(r1)→f1
mulf f0,f1→f2
stf f2→X(r1)
addi r1,4→r1
blt r1,r2,0
```



```
ldf X-4(r1)→f1
mulf f0,f1→f2
stf f2→X(r1)
mulf f0,f2→f3
stf f3→X+4(r1)
addi r1,4→r1
blt r1,r2,0
```

- Two mulf's are not parallel

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

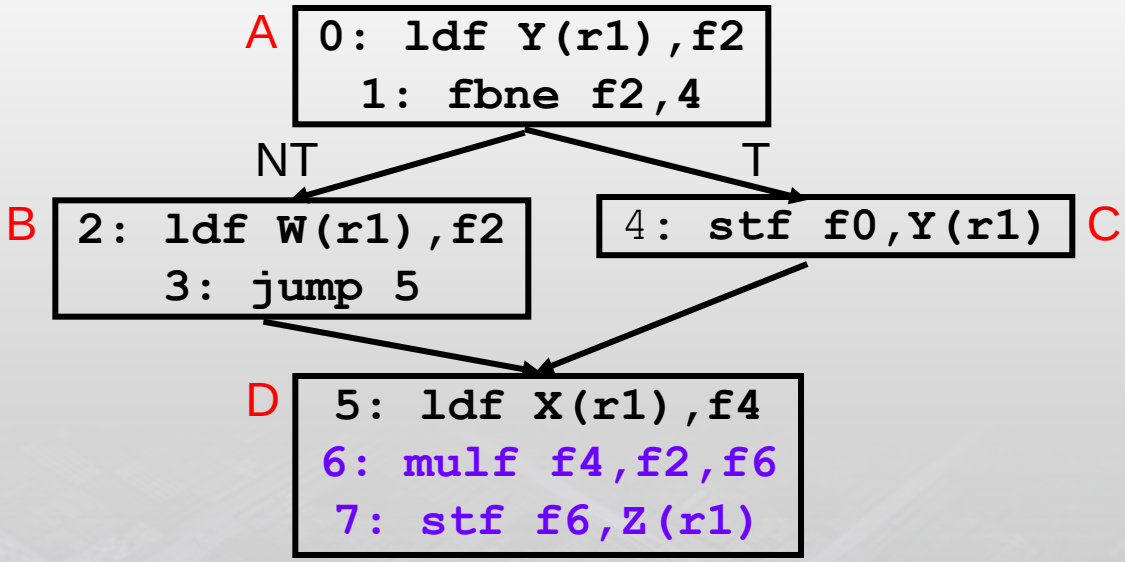
Source code

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

Machine code

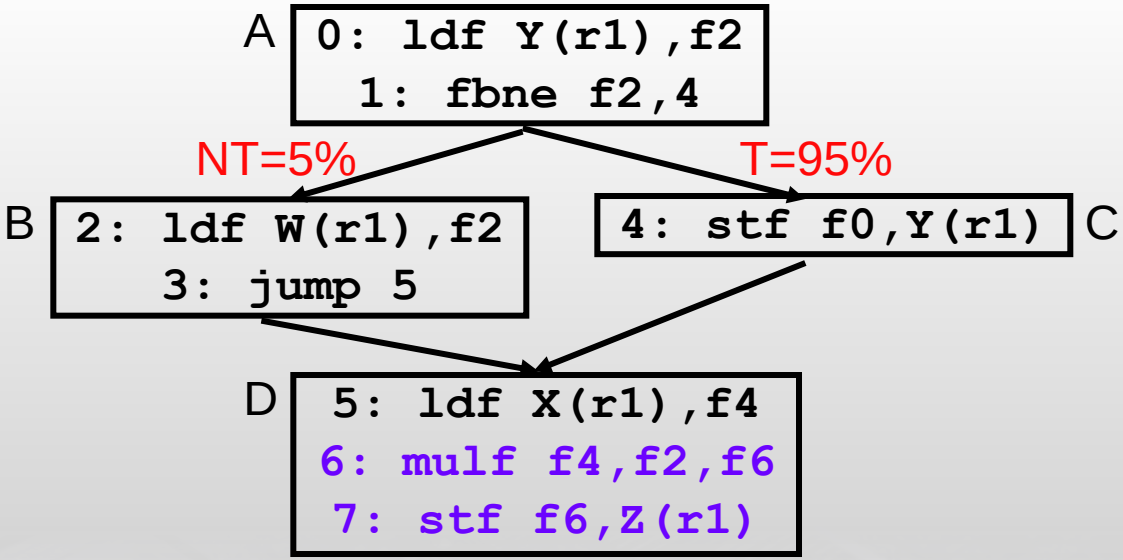
```
0: ldf Y(r1),f2
1: fbne f2,4
2: ldf W(r1),f2
3: jump 5
4: stf f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

4 basic blocks: A,B,C,D



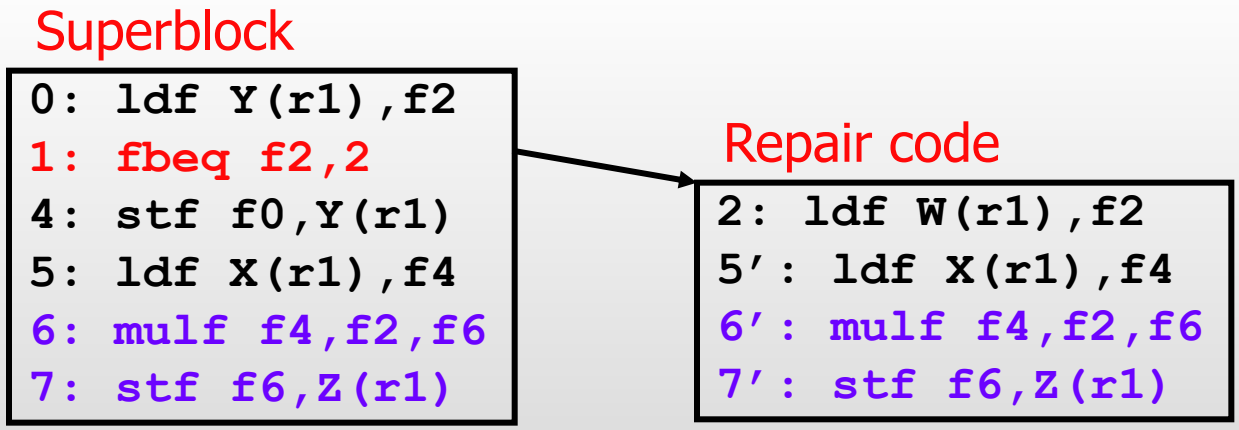
- Problem: separate #6 (3 cycles) from #7
- How to move **mulf** above if-then-else?
- How to move **ldf**?

Superblocks



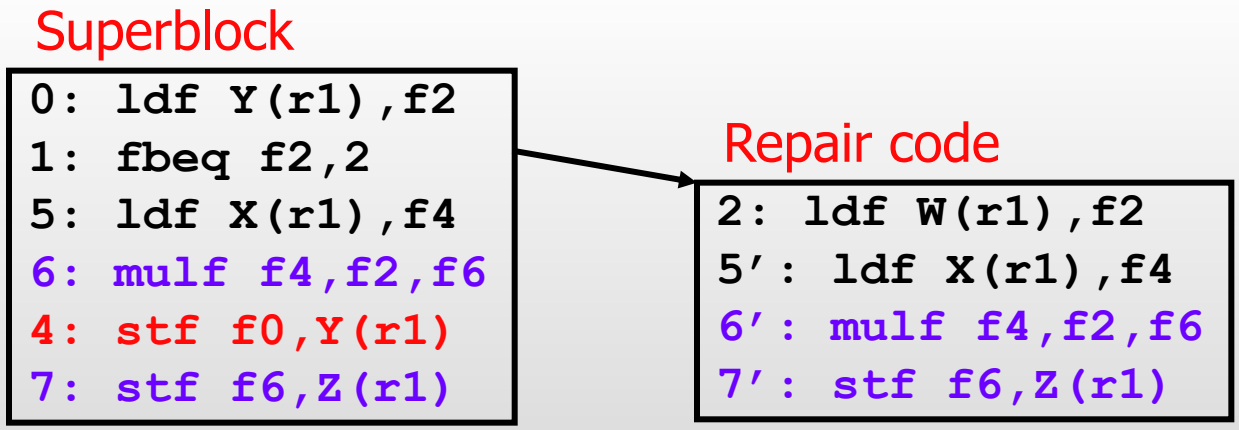
- First trace scheduling construct: **superblock**
  - Use when branch is highly biased
  - Fuse blocks from most frequent path: A,C,D
  - Schedule
  - Create **repair code** in case real path was A,B,D

Superblock and Repair Code



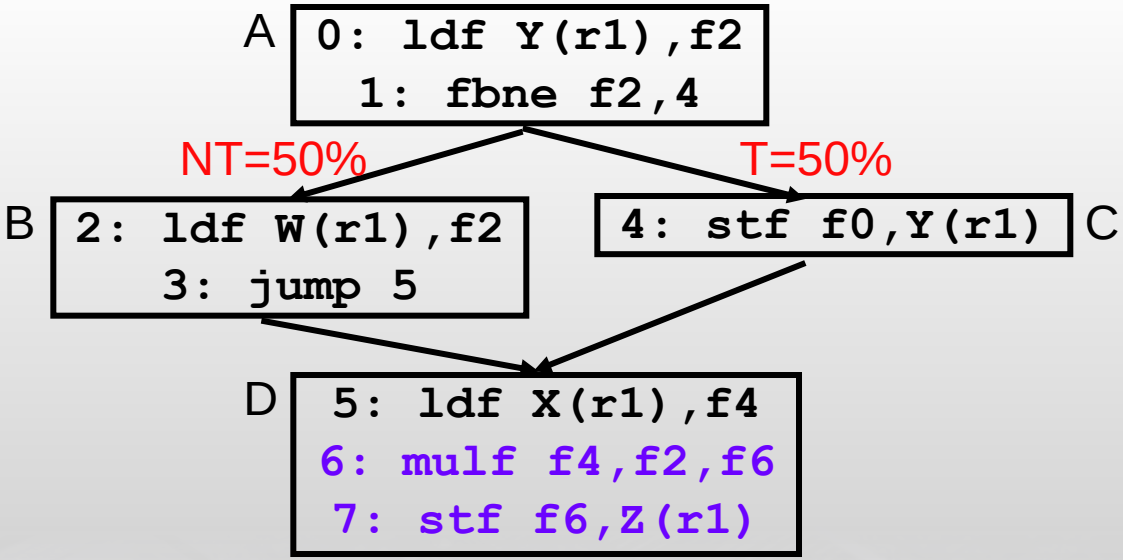
- What did we do?
  - Change sense (test) of branch 1
    - Original taken target now fall-thru
  - Created repair block
    - May need to duplicate some code (here basic-block D)
  - Haven't actually scheduled superblock yet

Superblocks Scheduling I



- 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

Non-Biased Branches: Use Predication



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

Hyper-block

```
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

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

Compiler Scheduling Redux

+ Can do some things with simple inner-loops
– Non-loop code is much more difficult
  - Basic-block ILP typically < 2
– Cache misses are a problem too

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

- And this is assuming pipeline will not block until f1 is needed
- If pipeline stalls immediately, performance will be even worse

Scheduling: Compiler or Hardware

- Compiler
  - \+ Potentially large scheduling scope (full program)
  - \+ Simple hardware → fast clock, short pipeline, and low power
  - – Low branch prediction accuracy (profiling?)
  - – Little information on memory dependences (profiling?)
  - – Can't dynamically respond to cache misses
  - – Pain to speculate and recover from mis-speculation (h/w support?)
- Hardware
  - \+ High branch prediction accuracy
  - \+ Dynamic information about memory dependences
  - \+ Can respnd to cache misses
  - \+ Easy to speculate and recover from mis-speculation
  - – Finite buffering resources fundamentally limit scheduling scope
  - – Scheduling machinery adds pipeline stages and consumes power

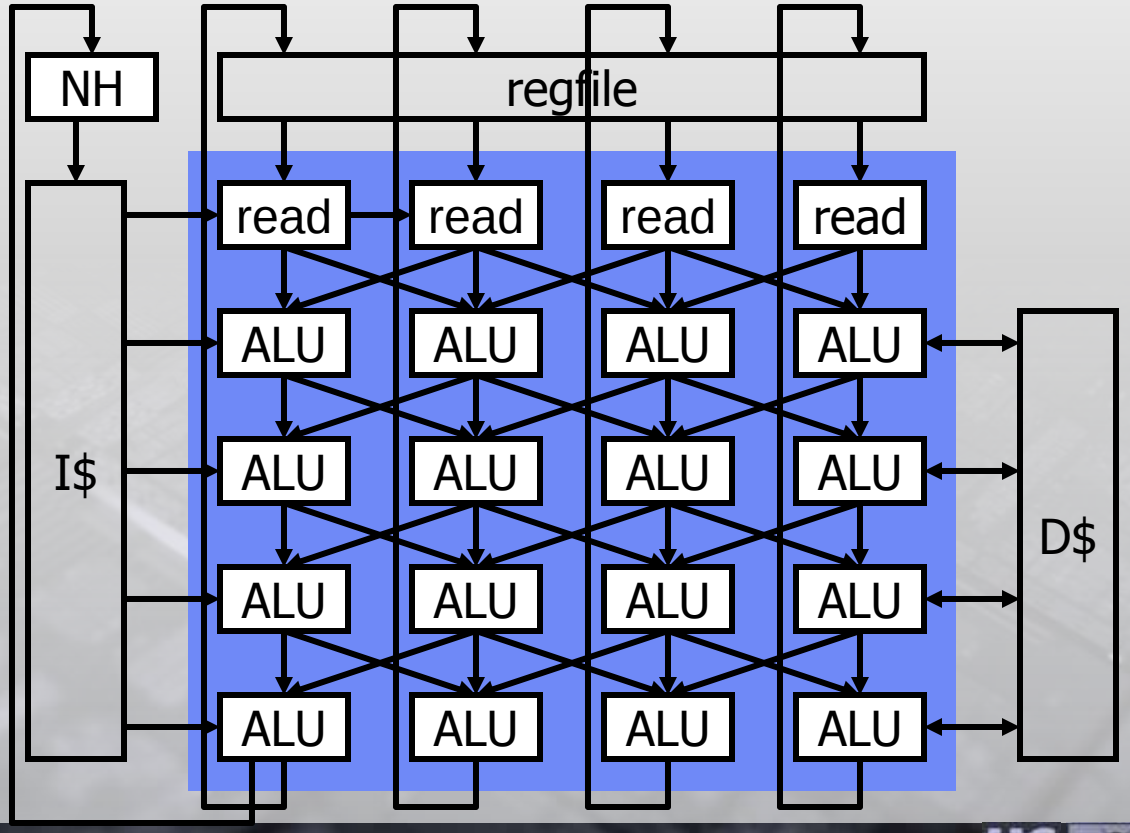
Research Topic

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 horizontal groups
  - Executes atomic code blocks (hyperblocks)
    - Uses predication and special scheduling to avoid taken branches
  - UT-Austin research project
    - http://www.cs.utexas.edu/users/cart/trips/
  - Fabricated an actual chip with help from IBM: next-generation PowerPC ¿?

Grid Processor

- Components
  - next hyperblock 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
```

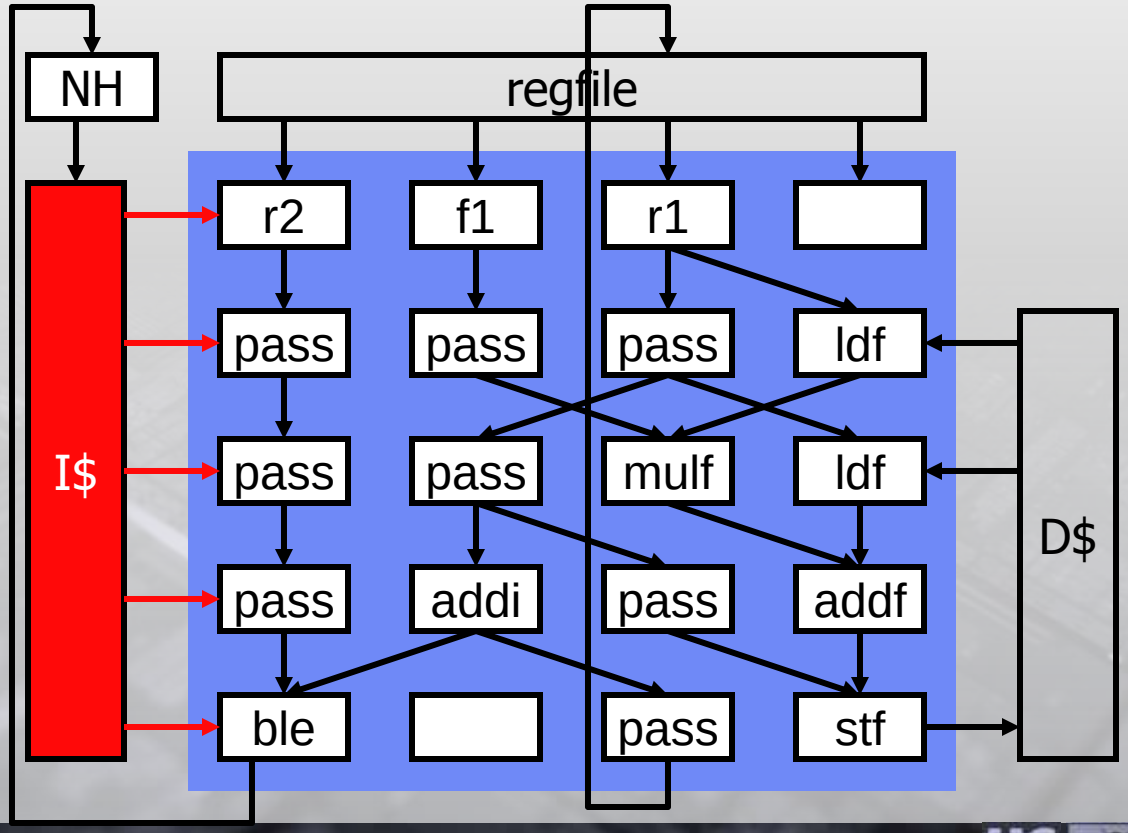
Grid Processor SAXPY

```
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

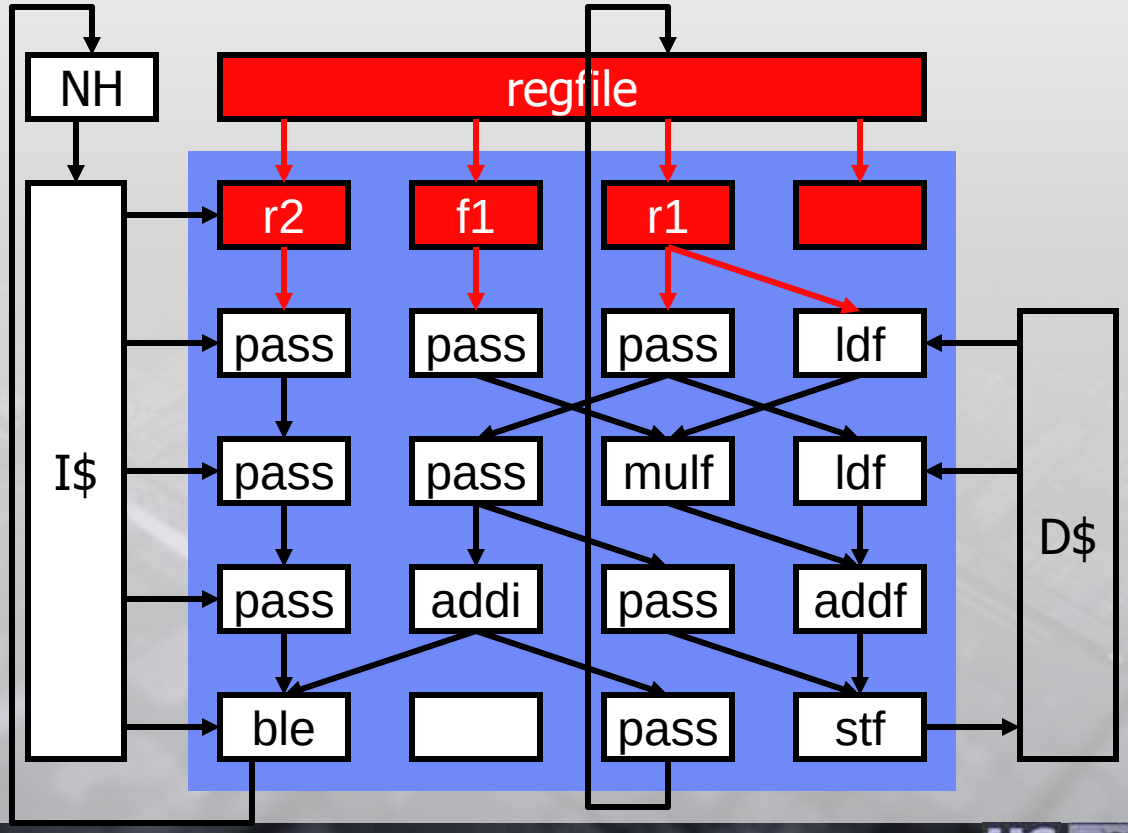
Grid Processor SAXPY Cycle 1

- Map code block to grid



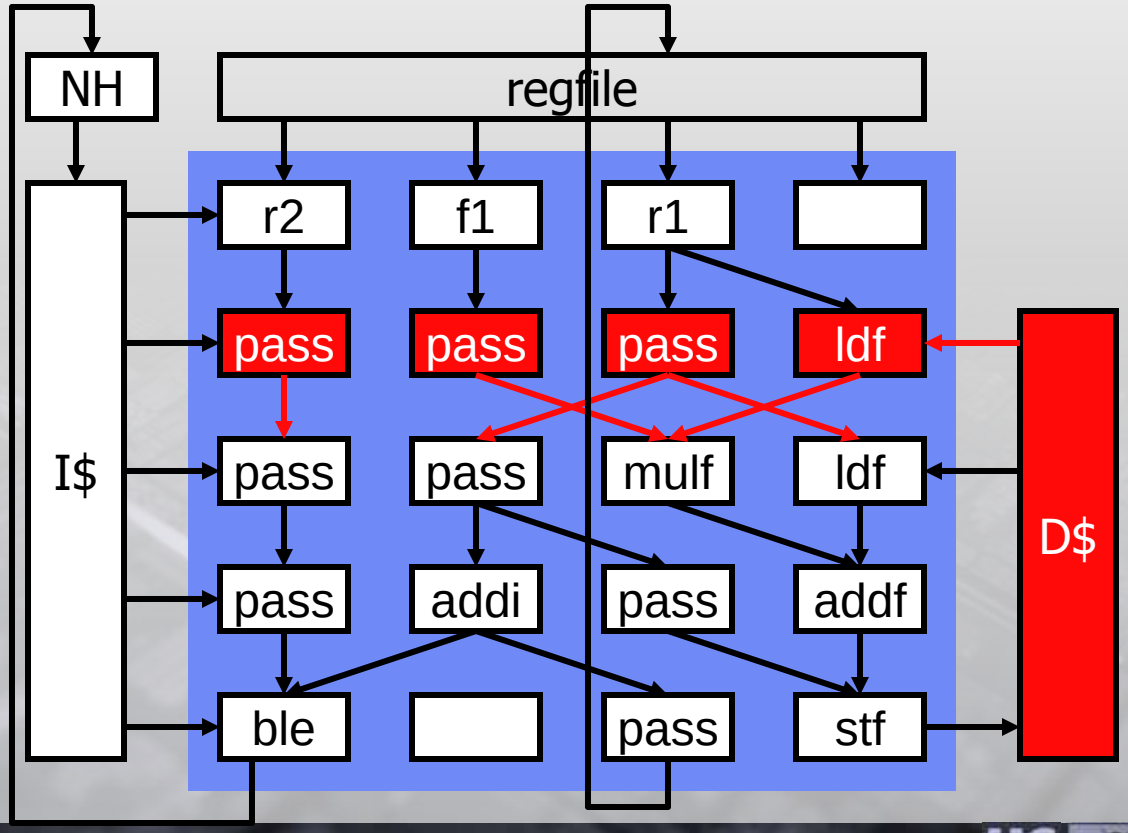
Grid Processor SAXPY Cycle 2

- Read registers



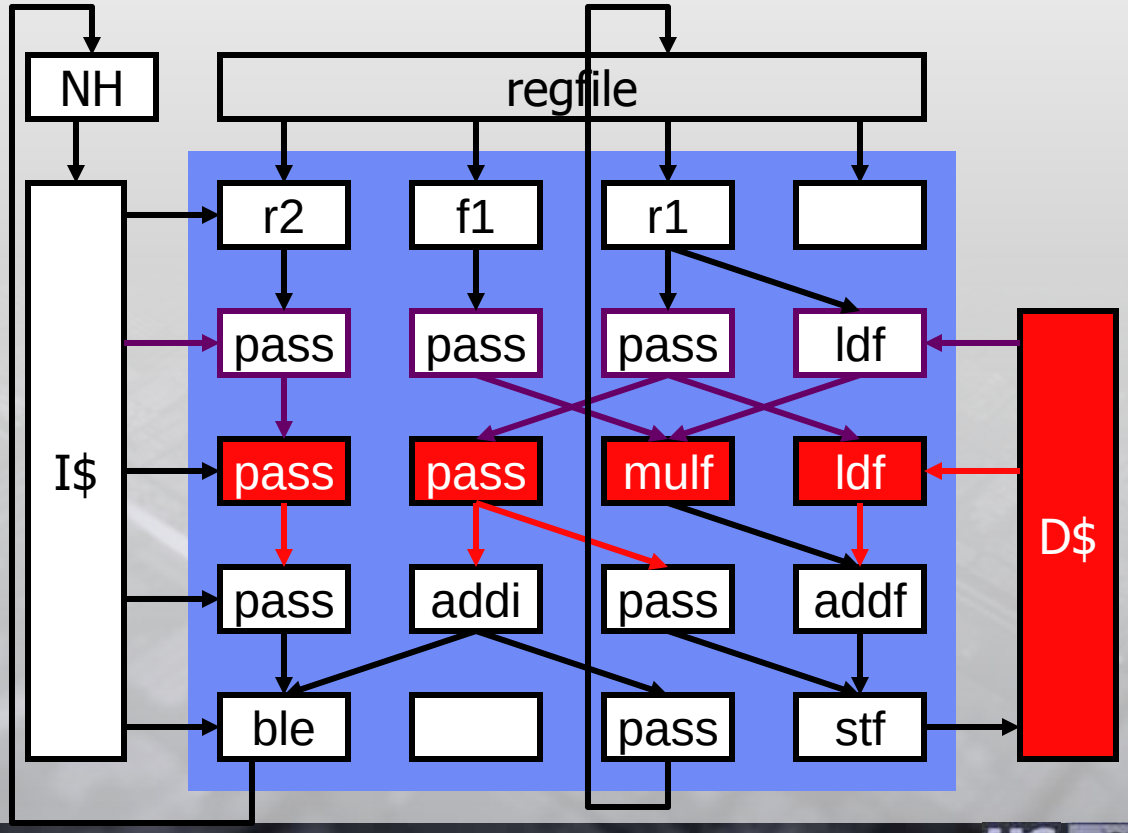
Grid Processor SAXPY Cycle 3

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



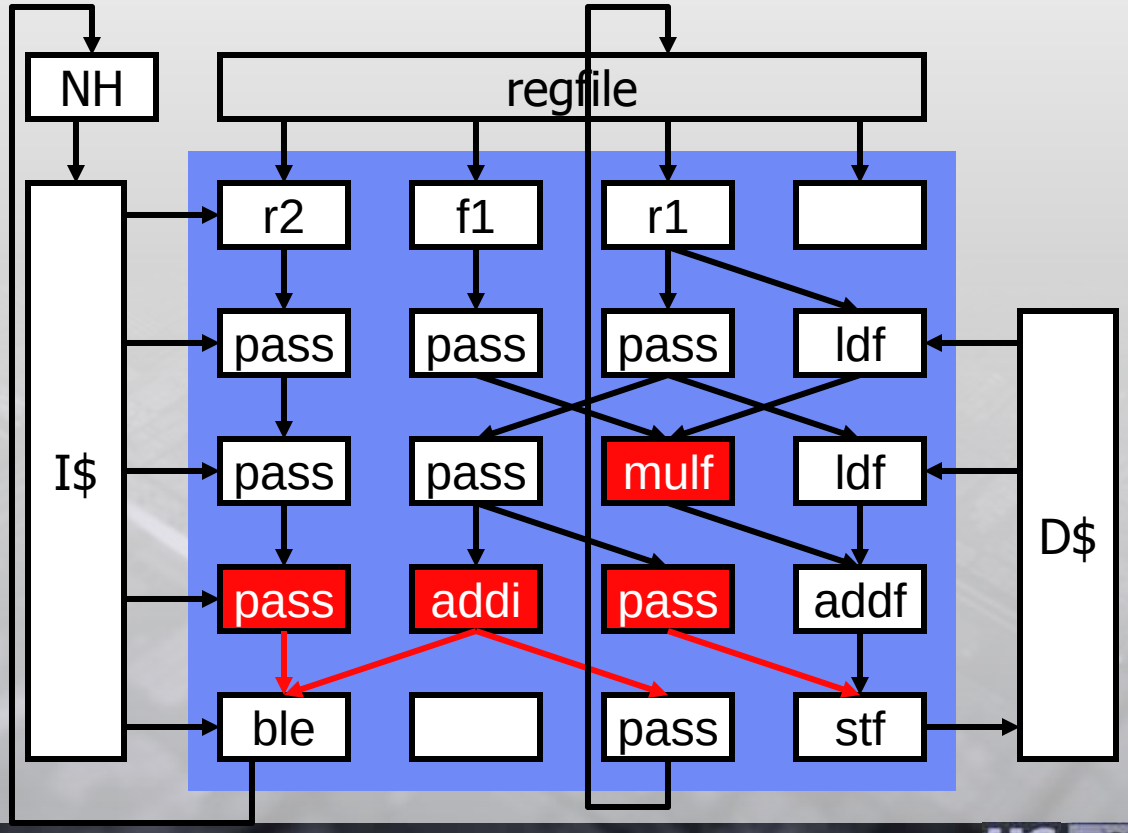
Grid Processor SAXPY Cycle 4

- Execute second grid row



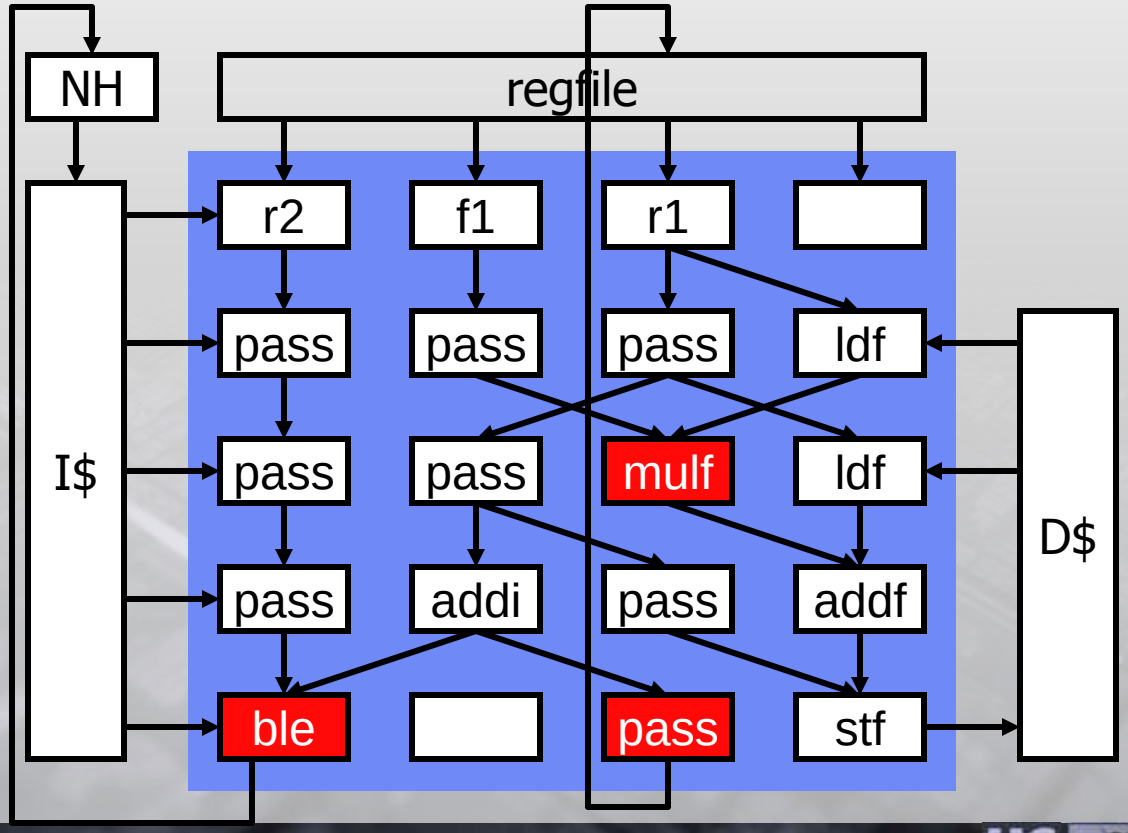
Grid Processor SAXPY Cycle 5

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



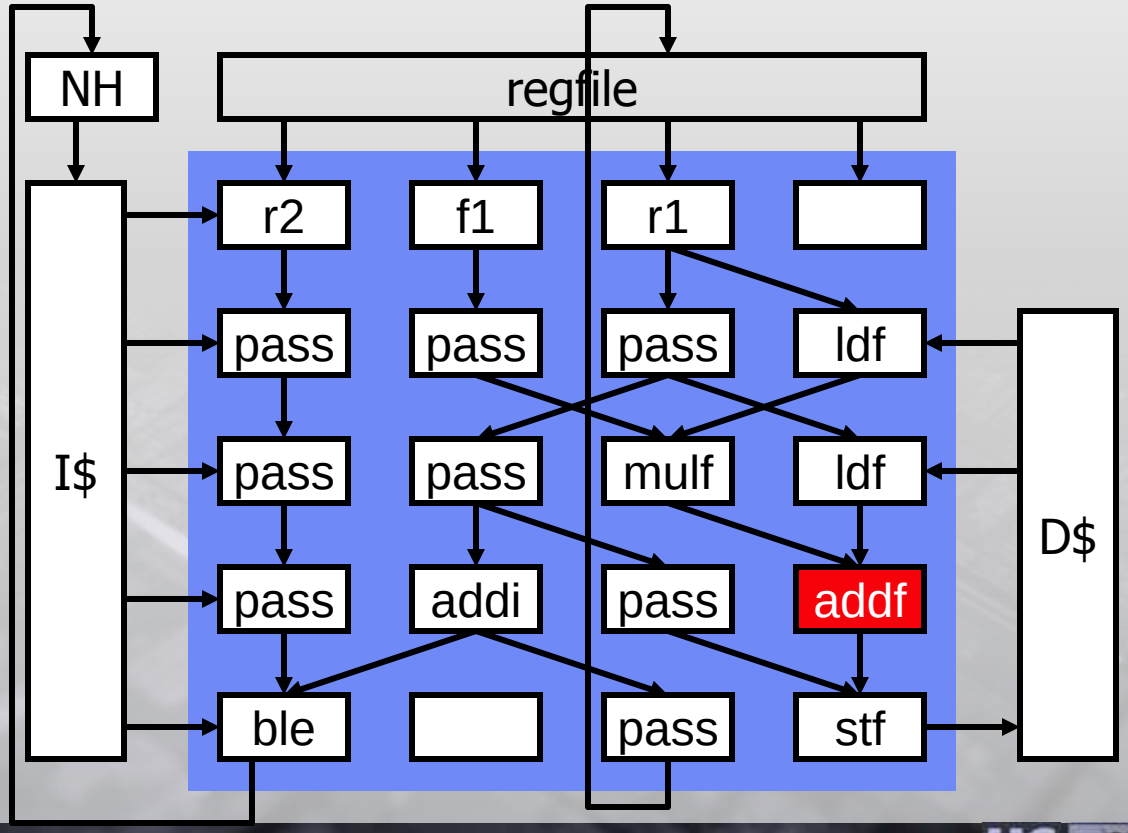
Grid Processor SAXPY Cycle 6

- Execute third grid row



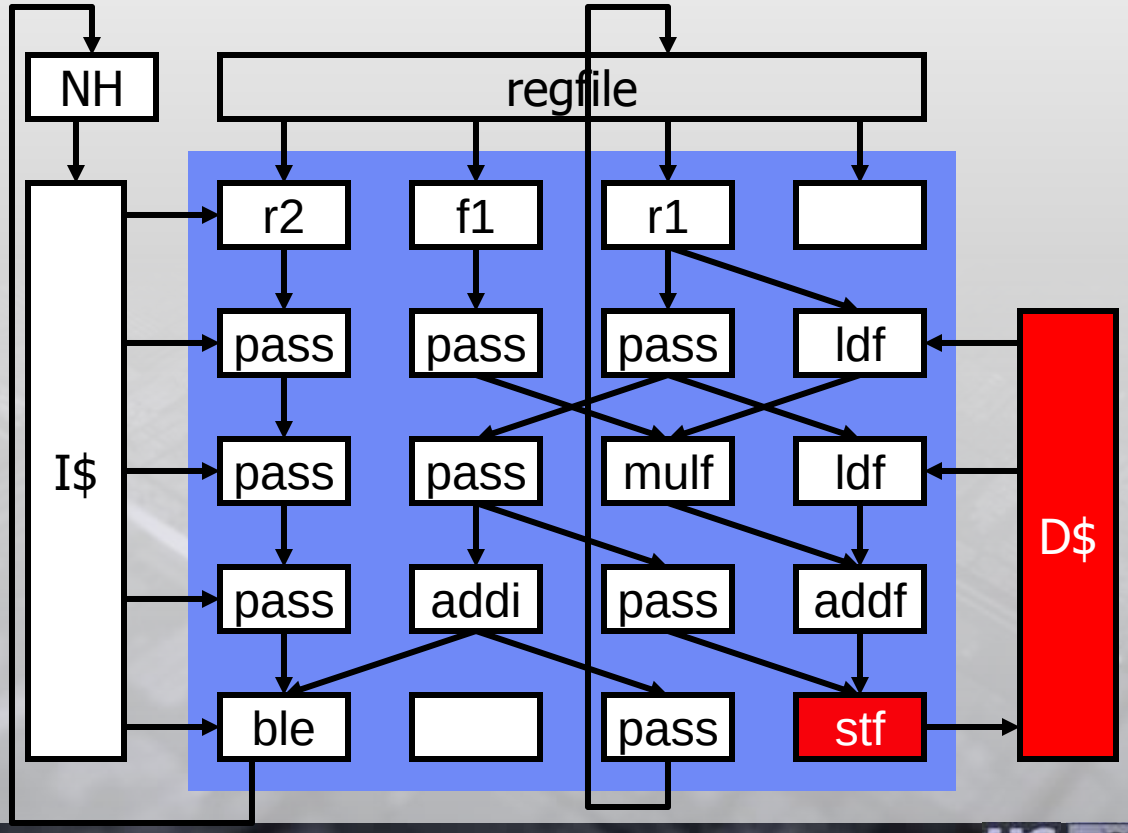
Grid Processor SAXPY Cycle 9

- Finish 1



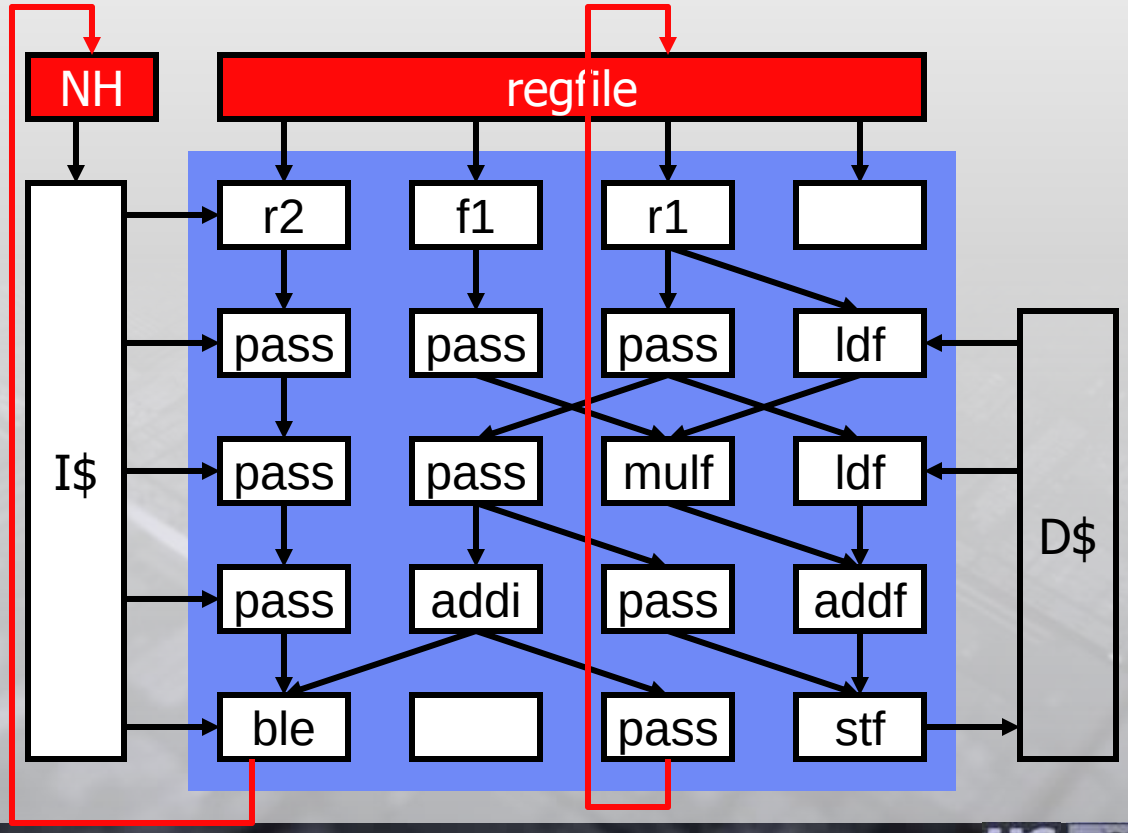
Grid Processor SAXPY Cycle 10

- Finish 2



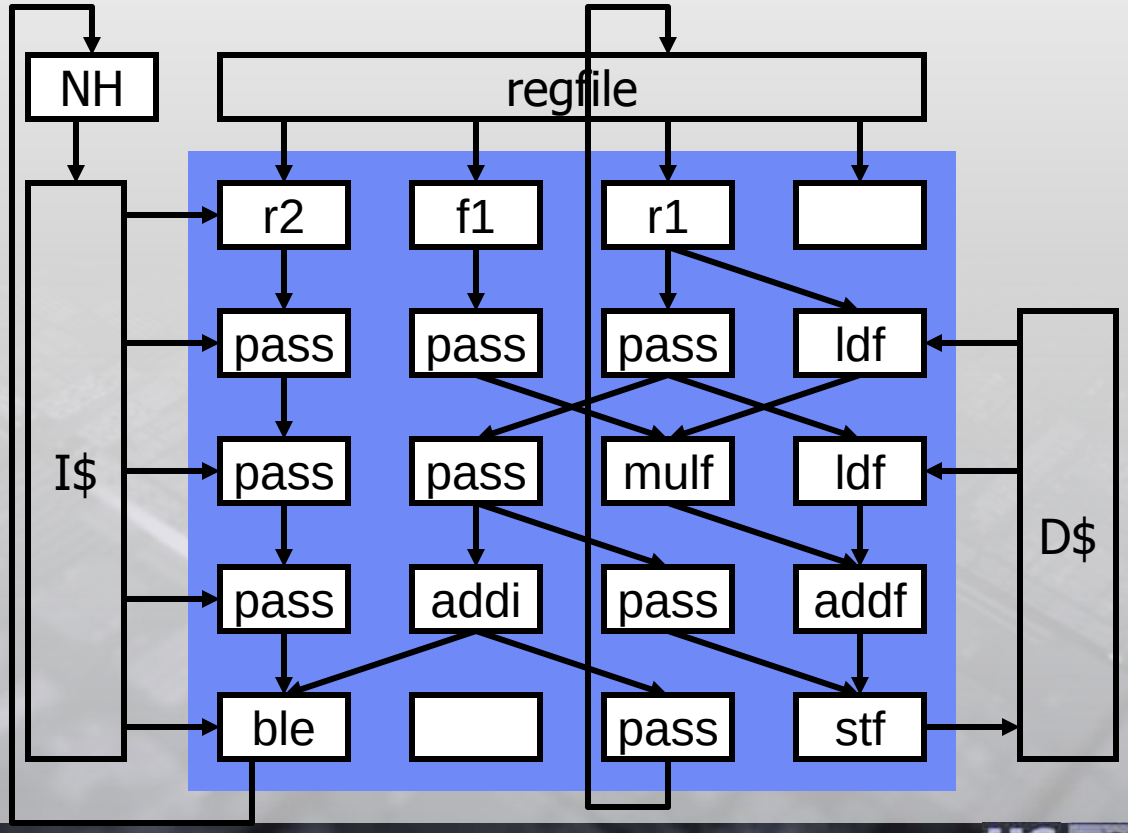
Grid Processor SAXPY cycle 11

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



Grid Processor SAXPY Performance

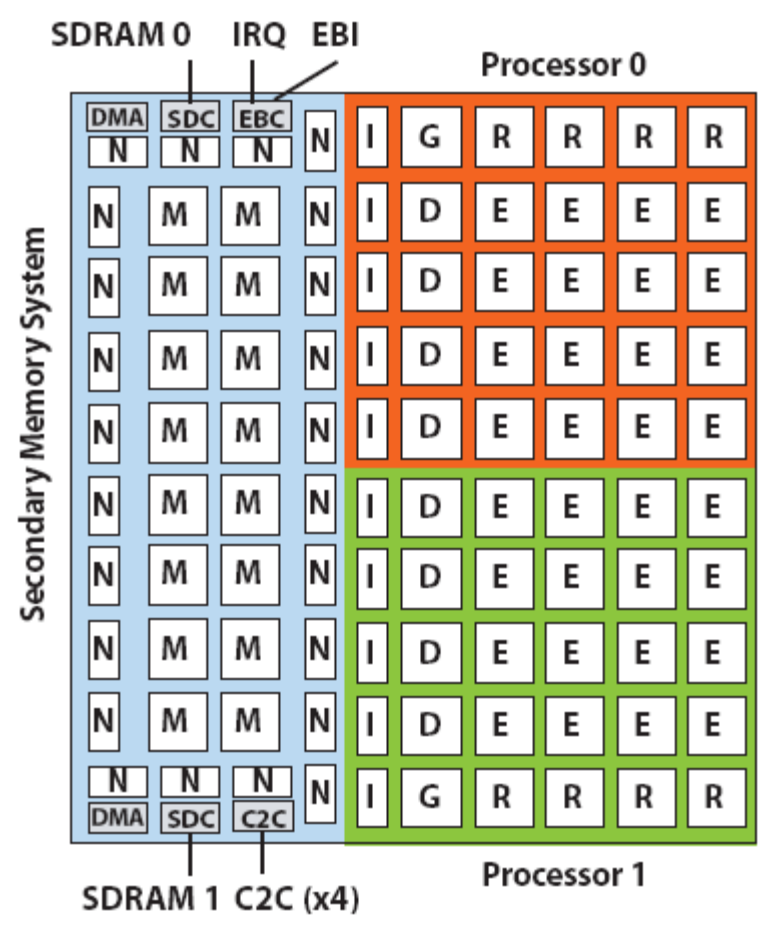
- Performance
  - 1 cycle fetch
  - 1 cycle read regs
  - 8 cycles execute
  - 1 cycle write regs
  - 11 cycles total
- ***Utilization***
  - 7 / (11 * 16) = 4%
- What's the point?
  - \+ Simpler components
  - \+ Faster clock?

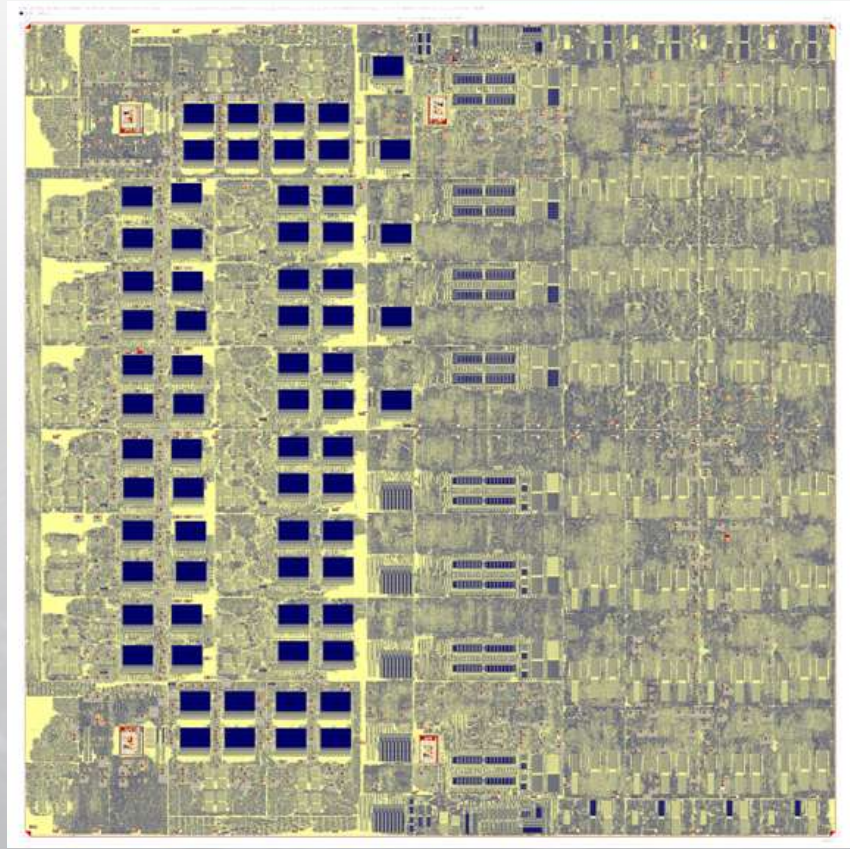


Grid Processor Redux

+ 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

Prototype


SDRAM 0
IRQ
EBI
Processor 0
DMA
SDC
EBC
C2C
SDRAM 1
C2C (x4)
Processor 1
Secondary Memory System
N
M
I
G
D
E
R



Acknowledgments

- Slides developed by Amir Roth of University of Pennsylvania with sources that included University of Wisconsin slides by Mark Hill, Guri Sohi, Jim Smith, and David Wood.
- Slides enhanced by Milo Martin and Mark Hill with sources that included Profs. Asanovic, Falsafi, Hoe, Lipasti, Shen, Smith, Sohi, Vijaykumar, and Wood
- Slides re-enhanced by V. Puente of University of Cantabria