National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 9-1: SuperScalar Processor

CS10014 Computer Organization

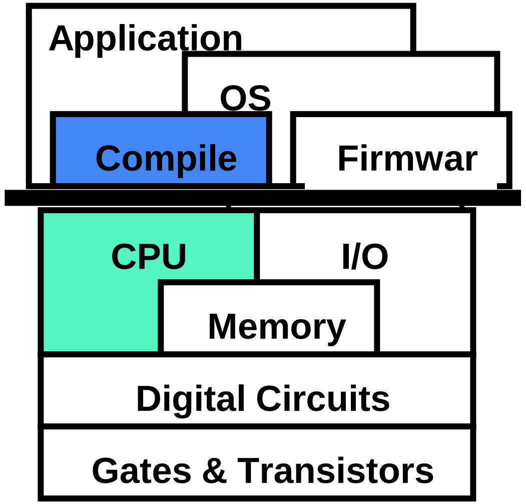
Department of Computer Science
Tsung Tai Yeh
Thursday: 1:20 pm– 3:10 pm
Classroom: EC-022

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CIS510 at Upenn
    - https://www.cis.upenn.edu/~cis5710/spring2019/
  - CSCE 513 at University of South Carolina
    - https://passlab.github.io/CSCE513/

Superscalar Execution

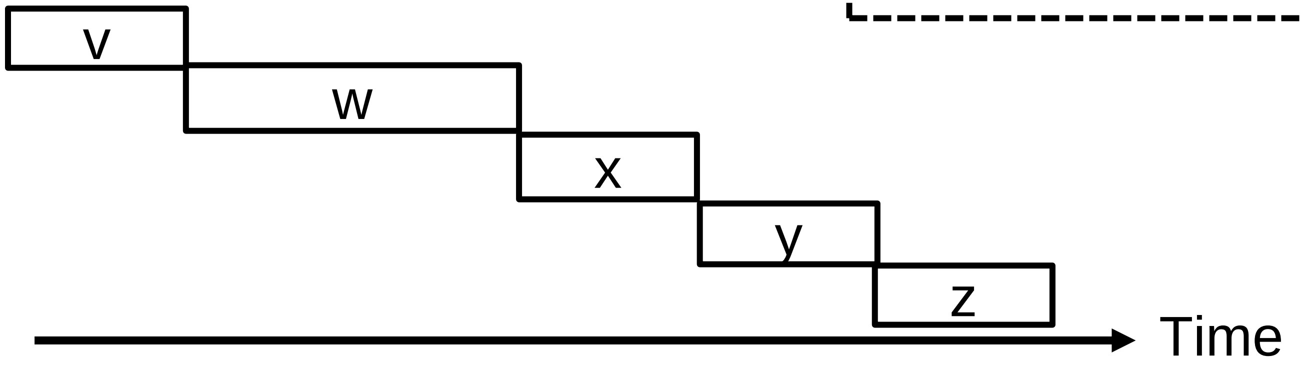
- Idea of instruction-level parallelism
- Superscalar scaling issues
  - Multiple fetch and branch prediction
  - Dependence-checks & stall logic
  - Wide bypassing
  - Register file & cache bandwidth
- Superscalar vs. VLIW/EPIC



In-order Execution

- Assumptions:
  - Single FP adder takes 2 cycles
  - Single FP multiplier takes 5 cycles
  - Can issue add & multiply together
  - Single adder

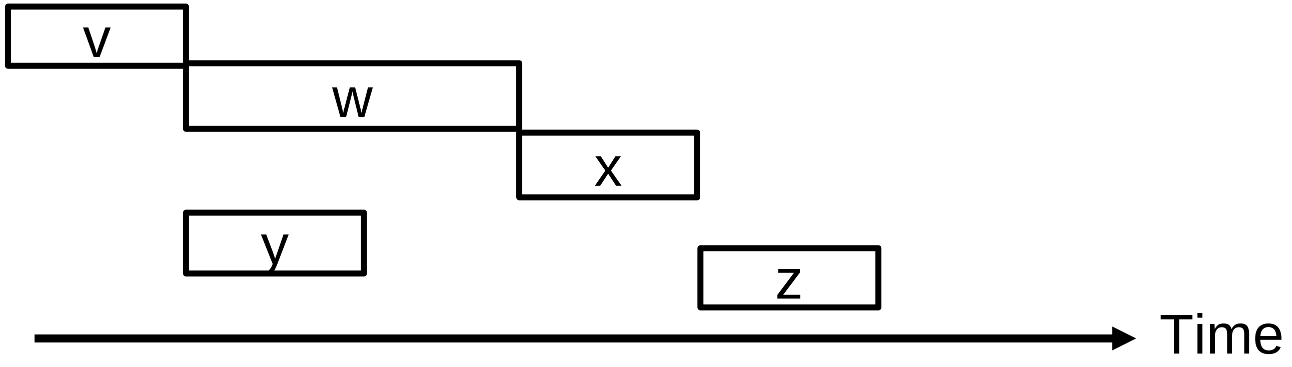
V:	add	$f2,	$f4,	$f10
W:	mul	$f10,	$f6,	$f10
X:	add	$f10,	$f8,	$f12
Y:	add	$f4,	$f6,	$f4
Z:	add	$f4,	$f8,	$f10



Out of Order Execution

- Can start Y as soon as adder available
- Z can be issued until $f10 is not busy and adder is available

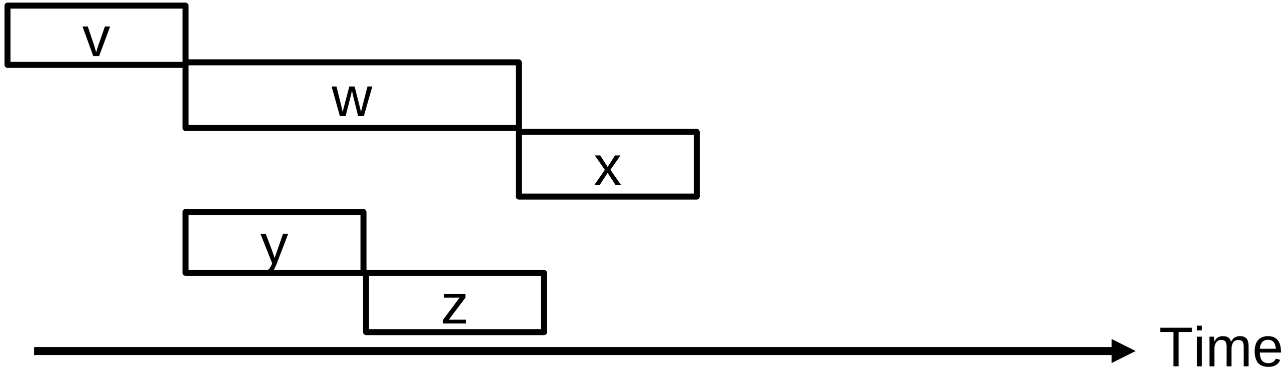
V:	add	$f2,	$f4,	$f10
W:	mul	$f10,	$f6,	$f10
X:	add	$f10,	$f8,	$f12
Y:	add	$f4,	$f6,	$f4
Z:	add	$f4,	$f8,	$f10



Register Renaming

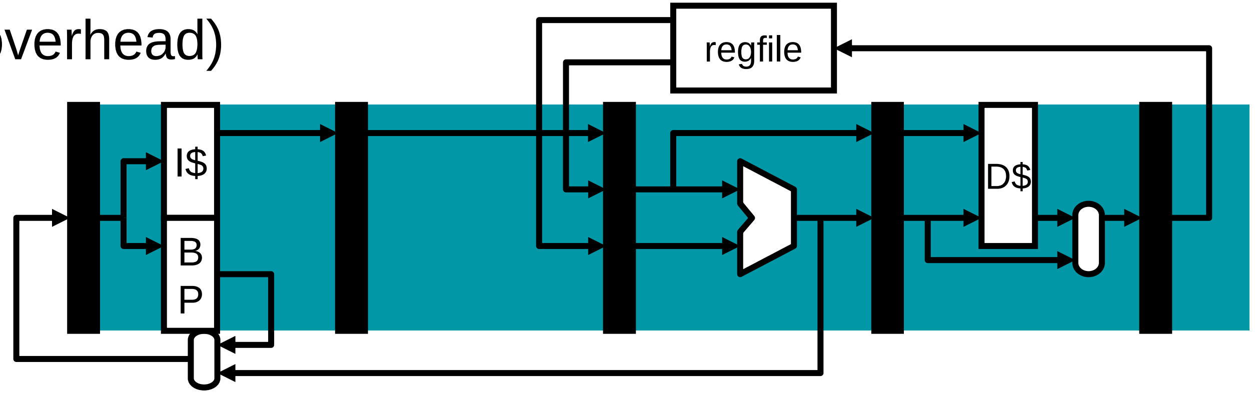
```
V:      add  $f2,  $f4,  $f10
W:      mul  $f10, $f6,  $f10
X:      add  $f10, $f8,  $f12
Y:      add  $f4,  $f6,  $f4
Z:      add  $f4,  $f8,  $f10
```

```
V:      add  $f2,  $f4,  $f10a
W:      mul  $f10a, $f6, $f10a
X:      add  $f10a, $f8, $f12
Y:      add  $f4, $f6,   $f4
Z:      add  $f4, $f8,   $f14
```



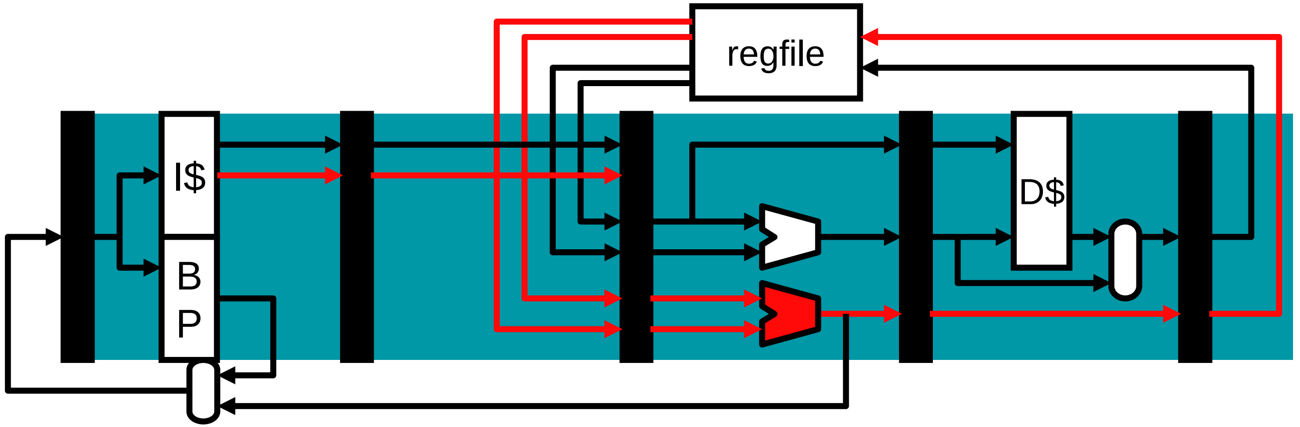
Scalar Pipeline and the Flynn Bottleneck

- **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]

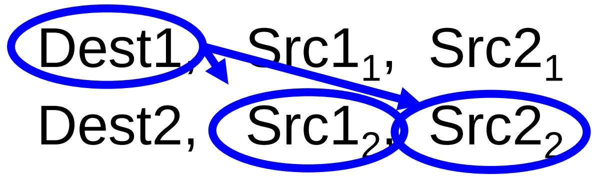


An Opportunity …

- Why not execute following two instructions **at the same time** ?
  - ADD R3, R1, R2
  - ADD R6, R4, R5
- What about ?
  - ADD R3, R1, R2
  - ADD R6, R4, R3
  - In this case, dependencies prevent the parallel execution
- Could we issue more instructions at a time ?

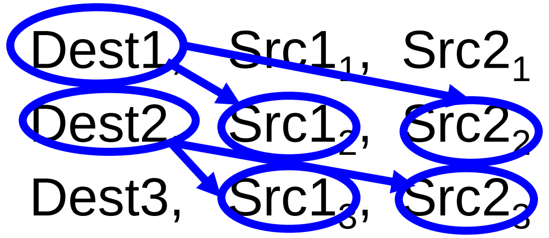
Problem: Checking Dependency

- For two instructions: **2 checks**



  - ADD Dest1, $Src1_1$, $Src2_1$
  - ADD Dest2, $Src1_2$, $Src2_2$

- For three instructions: **6 checks**

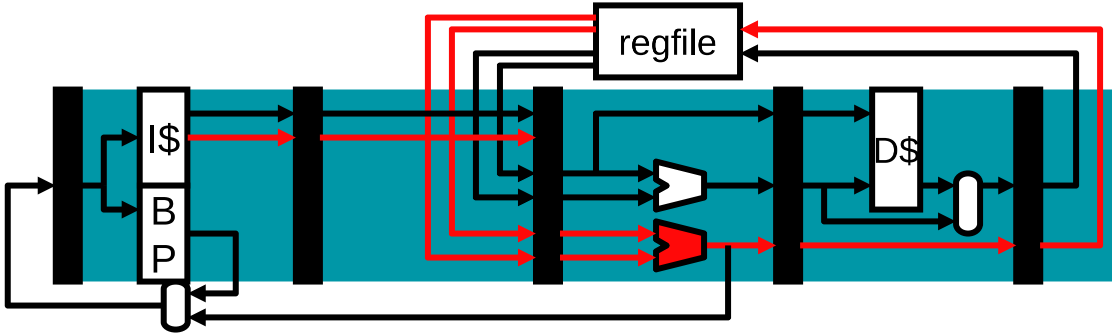


  - ADD Dest1, $Src1_1$, $Src2_1$
  - ADD Dest2, $Src1_2$, $Src2_2$
  - ADD Dest3, $Src1_3$, $Src2_3$

- **How many checks needed in four instructions ?**
- Plus checking for load-to-use stalls for prior n loads

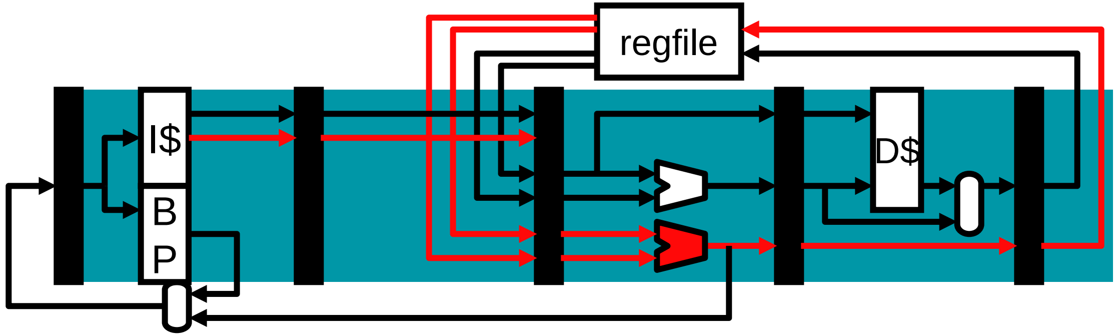
Simple Dual-issue Pipeline

- 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



Superscalar Pipeline Diagrams -- Ideal

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 r12,r13→r6`					F	D	X	M	W			
`add r14,r16→r7`						F	D	X	M	W		
`add r15,r17→r8`							F	D	X	M	W	
`ld [r18]→r9`								F	D	X	M	W

2-way superscalar	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 r12,r13→r6`			F	D	X	M	W				
`add r14,r16→r7`			F	D	X	M	W				
`add r15,r17→r8`				F	D	X	M	W			
`ld [r18]→r9`				F	D	X	M	W			

Superscalar Pipeline Diagrams -- Realistic

Scalar

	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					
`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	
`ld [r8]→r9`								F	D	X	M	W

2-way superscalar

	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					
`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	**d***	X	M	W		
`ld [r8]→r9`					F	D	X	**d***	M	W	

Superscalar is only one cycle fast

Test Yourself !!

- How many cycle is spent in superscalar pipelining ?

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`											
`add r4,r6➔r7`											
`add r5,r7➔r8`											
`ld [r8]➔r9`											

Test Yourself !!

- How many cycle is spent in superscalar pipelining ?
  - 10 cycles

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
- **Wide instruction decode**
  - Replicate decoders
- **Wide instruction issue**
  - Determine when instructions can proceed in parallel
  - More complex stall logic - order N^2 for *N*-wide machine
  - Not all combinations possible
- **Wide register read**
  - One port for each register read
  - Example, 4-wide superscalar ➔ >= 8 read ports
  - Each port needs its own set of address and data wires

Superscalar Challenges - Back End

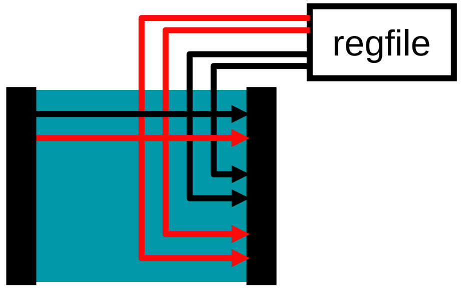
- **Wide instruction execution**
  - Replicate arithmetic units
  - Multiple cache ports (slower access, higher energy)
- **Wide instruction register writeback**
  - One write port per instruction that writes a register
  - Example, 4-wide superscalar ➔ >= 4 write ports
- **Wide bypass paths**
  - More 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 stall
    - Even for scalar machines (to fill load-use delay slot)
    - Even harder to schedule multiple-issue (superscalar)
- How much ILP is common ?
    - IPC (or CPI) depends on the applications
- Even given unbounded ILP, superscalar has implementation limits
    - IPC (or CPI) vs clock frequency trade-off
    - Given these challenges, what is reasonable today ?
        - ~4 instruction per cycle maximum

Superscalar Decode & Register Read

- 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 read + N write (2 read + 1 write per insn) (latency ∝ #ports)
    + Actually less than 2N, most values come from bypasses



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
`ld r4,0(r1)`	F	D	X	M	W
`addi r4,1,r4`	F	D	**d***	**d***	X
`sub r3,r5,r2`		F	**p***	**p***	D
`st r3,0(r1)`		F	**p***	**p***	D
`ld r8,4(r1)`					F

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

N^2 Dependence Cross-Check

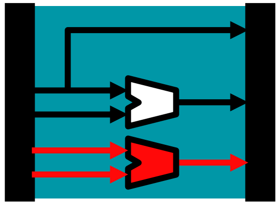
- Stall logic for 1-wide pipeline with full bypassing
  - Full bypassing = load/use stalls only

    X/M.op==LOAD && (D/X.rs1==X/M.rd || D/X.rs2==X/M.rd)
  - Two "terms": $\propto$ 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": $\propto 2N^2$
    - This is the **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

Superscalar Execute

- What is involved in executing multiple (N) instructions 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 instruction mix
    - Intel Pentium: 1 any + 1 ALU



D$ Bandwidth: Multi-Porting, Replication

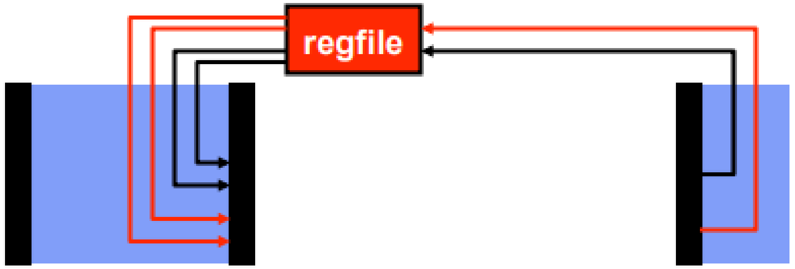
- How to provide additional D$ (D-cache) bandwidth ?
  - Have already seen split I$/D$, but that gives you just one D$ port
  - How to provide a second (maybe even a third) D$ port ?
- Option#1: **multi-porting**
  - + Most general solution, any two accesses per cycle
  - - Lots of wires; expansive in latency, area (cost), and power
- Option#2: **replication**
  - Read from either replica, but writes update both replicas
    - Writing both insures they have the same values
  - Multiplies read bandwidth only (writes must go to all replicas)
  - + General solution for loads, little latency penalty
  - - Not a solution for stores, area, power penalty

D$ Bandwidth: Banking

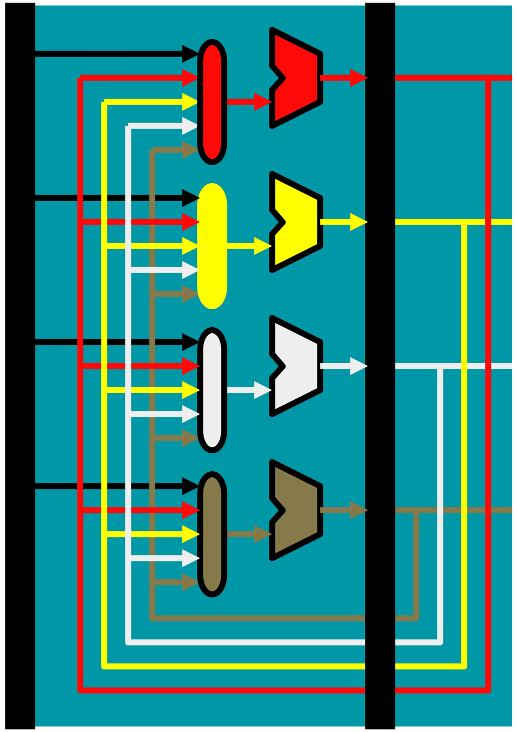
- Option#3: **banking** (or **interleaving**)
  - Divide D$ into banks (by address), one access per bank per cycle
  - **Bank conflict**: two access to same bank -> one stall
  - + No latency, area, power overheads
  - + One access per bank per cycle, **assuming no conflicts**
  - - Complex stall logic -> 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 missing handling
  - Index bits? Banks contain complete cache lines
    - - More conflicts
    - + Tags not replicated, simpler missing handling

Superscalar Register Read/Write

- How many register ports to execute N insns per cycle ?
  - 2N read + N write (2 read + 1 write per insn)
  - In reality, fewer than that
    - Read ports: some instructions read only one register
    - Write ports: store, branches (35 % insns) don't write registers
  - Multi-porting and replication both work for register files
  - Banking ? Not used (conflicts too hard to handle)



Superscalar Bypass



- **N^2 bypass network**
  - N+1 input muxes at each ALU input
  - N^2 point-to-point connections
  - Routing lengthens wires
  - Heavy capacitive load
  - Expensive metal layer crossings
- One of the big problems of superscalar
  - Why ? On the critical path of single-cycle "bypass & execute" loop

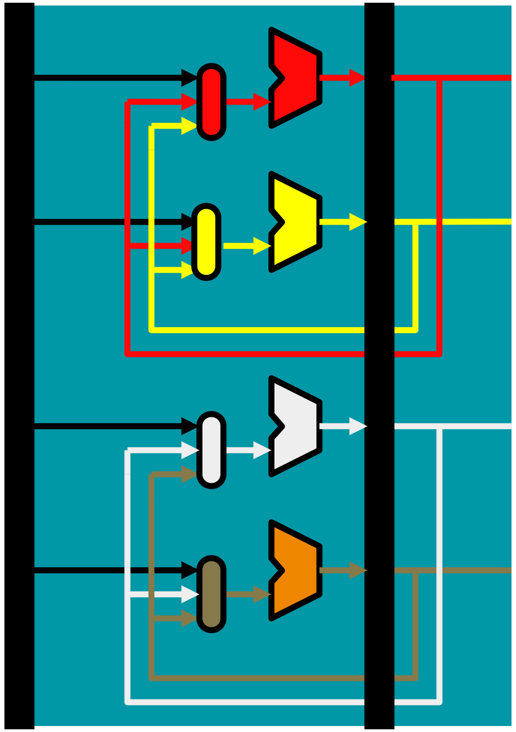
Not All N^2 Created Equal

- N^2 bypass vs. N^2 stall logic & dependence cross-check
- N^2 bypass
  - Multiple levels (MX, WX) of bypass
  - Must fit in one clock period with ALU
- Dependence cross-check not even 2nd biggest N^2 problem
  - Register file is also an N^2 problem (think latency where N is # ports)
  - And also more serious than cross check

Superscalar Register Bypass

- **Flow of data between instructions**
  - R1 = R3 * R4
  - R7 = R1 + R2
  - RAW (Read-after-Write) dependency occurs in R1
- **Register Bypassing**
  - Hardware mechanism allows data to flow directly from the output of one instruction to the input of another
  - The result of the first instruction is written to register R1
  - At the same time, a second copy of the result is **piped directly to the arithmetic unit** that consumes the value
  - Requires a hardware interconnection network between outputs of functional units (such as adders, multipliers) and the inputs of other functional units

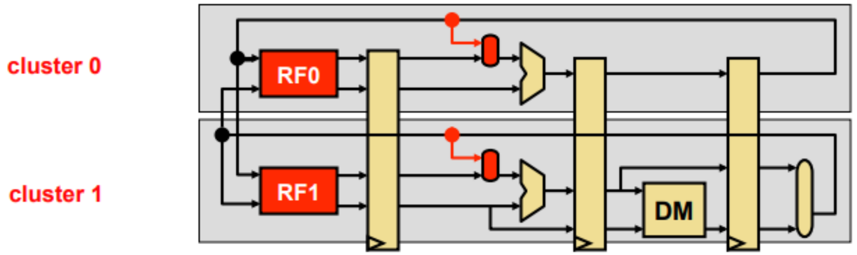
Mitigate N^2 Bypass : Clustering



- **Clustering**: mitigates N^2 bypass
  - Group ALUs into **K** clusters
  - Full bypassing within a cluster
  - Limited bypassing between clusters
    - With a one cycle delay
    - Can hurt IPC, but faster clock
  - (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
- **Cluster register file**, too
  - Replicate a register file per cluster
  - All register writes update all replicas
  - Few read ports; only for cluster

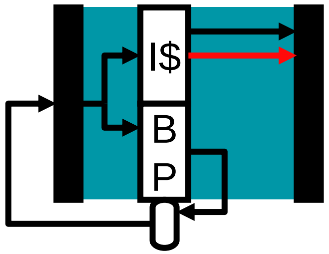
Mitigating N^2 RegFile: Clustering ++

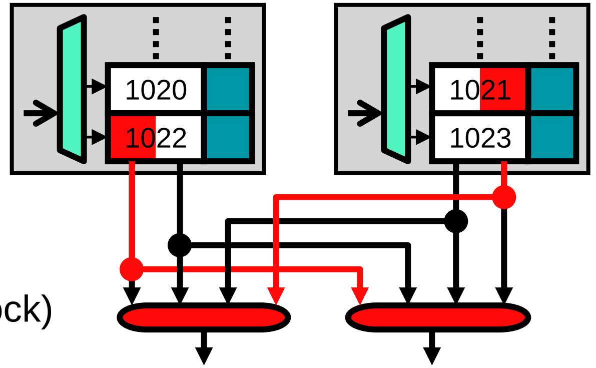
- Clustering: split N-wide execution pipeline into K clusters
  - With centralized register file, 2N read ports and N write ports
- Clustered register file: extend clustering to register file
  - Replicate the register file (one replica per cluster)
  - Register file supplies register operands to just its cluster
  - All register writes go to all register files (keep them in sync)
  - Advantage: fewer read ports per register
    - K register files, each with 2N/K read ports and N write ports



Superscalar Fetch

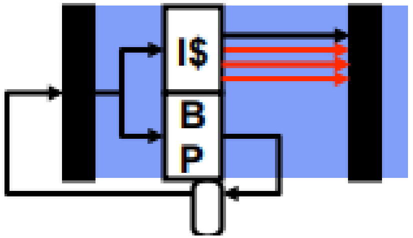
- What is involved in fetching multiple instructions per cycle?
- In same cache block? → no problem
  - 64-byte cache block is 16 instructions (~4 bytes per instruction)
  - Favors larger block size (independent of hit rate)
- Compilers align basic blocks to I$ lines (pad with **`nops`**)
  - – Reduces effective 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)





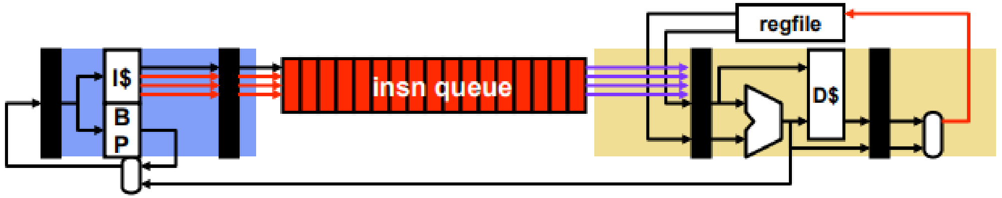
Limits of Simple Superscalar Fetch

- How many instructions can be fetched on average ?
  - BTB predicts the next block of instructions to fetch
    - Support multiple branch predictions per cycle
    - Discard post-branch insns after first branch predicted as “take
  - Lowers effective fetch width and IPC
  - What is the average number of instructions per taken branch ?
    - Assume: 20% branches, 50% taken -> ~10 instructions
  - Consider a 5-instruction loop with 4-issue processor
    - Without smarter fetch, ILP is limited to 2.5 (not 4)
  - Compiler could “unroll” the lop (reduce taken branches)
  - What else can we increase fetch rate ?



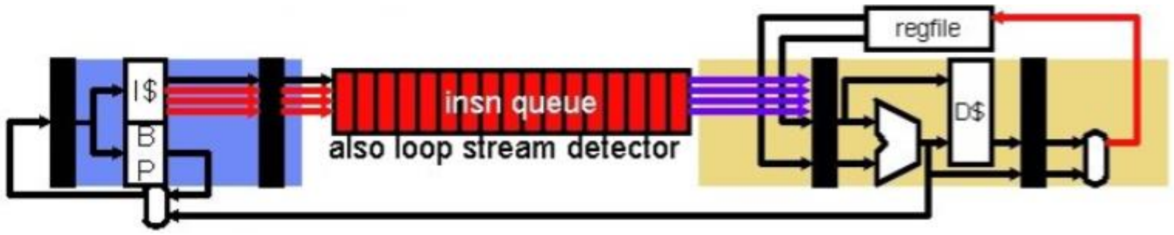
Increasing Superscalar Fetch Rate

- Option #1: over-fetch and buffer
  - Add a queue between fetch and decode (18 entries in Intel Core 2)
  - Compensates for cycles that fetch less than maximum instructions
  - “decouples” the “front end” (fetch) from the “back end” (execute)
- Option #2: predict next two blocks (extend BTB)
  - Transmits two PCs to fetch stage: “next PC” and “next-next PC”
  - Access I-cache twice (requires multiple ports or banks)
  - Requires extra merging logic to select and merge correct insns
  - Elongates pipeline, increase branch penalty



Increasing Superscalar Fetch Rate

- Option #3: Loop stream detector (Intel Core i7)
  - Put entire loop body into a small cache
  - Core i7: 28 micro-ops (avoids re-decoding macro-ops !)
  - Any branch mis-prediction requires normal re-fetch
- Option #4: trace cache (Pentium 4)
  - Tracks “traces” of disjoint but dynamically consecutive instructions
  - Pack (predicted) taken branch & its target into a one “trace” entry
  - Fetch entire “trace” while predicting the “next trace”



Impact of Branch Prediction

- Base CPI for scalar pipeline is 1
- **Base CPI for N-way superscalar pipeline is 1/N**
  - Amplifies stall penalties
  - Assume no data stalls (an overly optimistic assumption)
- Example: Branch penalty calculation
  - 20% branches, 75% taken, 2 cycle penalty, no 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
- How about 4-way superscalar pipeline slowdown?

Predication

- Branch mis-predictions hurt more on superscalar
    - Replace difficult branches with something else …
    - Convert control flow into data flow (& dependencies)
    - Avoids mis-predictions by removing hard-to-predict branches
    - Can hurt performance if branch was highly predictable
- **Predication**: insns conditionally executed
    - **Full predication** (ARM, Intel Itanium)
        - Can tag every insn with predicate, but extra bits in instruction
    - **Conditional moves** (Alpha, x86)
        - Construct appearance of full predication from one primitive
            - comoveq r1, r2, r3 // if (r1 == 0) r3 = r2
        - - Doesn't handle conditional memory operations
        - + Only good way of adding predication to an existing ISA

Predication

- Full predication vs partial predication
  - "T" means always executed
  - Full predication yields true (p2) and false (p3) predicate
  - In the partial-predication case, we use a "select" to implement "a = b ? c : d"

```
x = *ap;
if (x > 0)
    t = *bp + 1;
else
    t = x – 1;
*zp = t
```

Full predication

T	ld	t1 = 0[ap]
T	cmpgt	p2, p3 = t1, 0
p2	ld	t3 = 0[bp]
p2	add	t4 = t1, 1
p3	sub	t4 = t1, 1
T	Store	0[zp] = t4

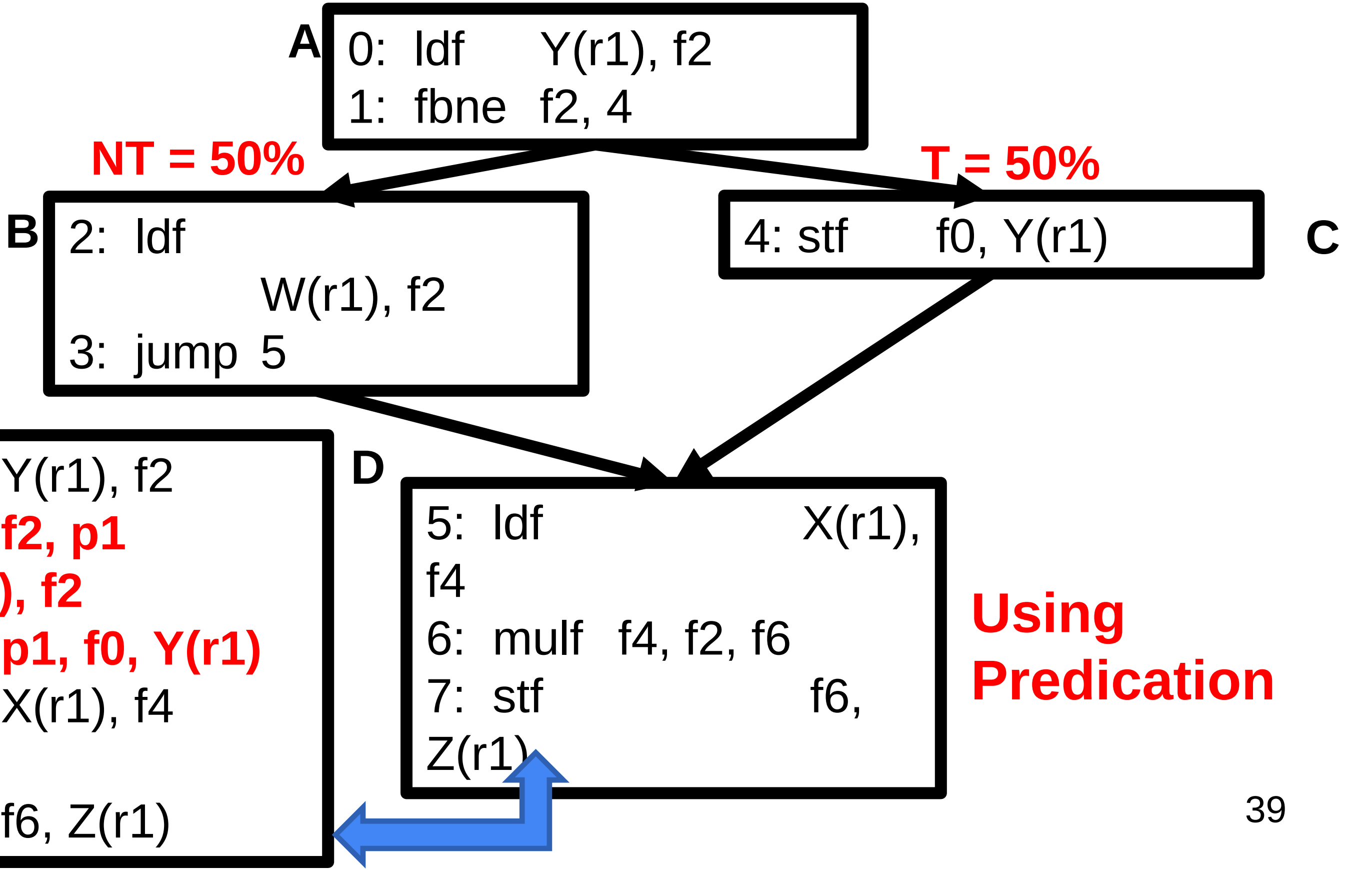
Partial predication

ld	t1 = 0[ap]
cmpgt	p2, p3 = t1, 0
ld	t3 = 0[bp]
add	t4 = t1, 1
sub	t4 = t1, 1
select	t6 = p2, t4, #5
store	0[zp] = t4

Predication If-Conversion Example

Source code

```
A = Y[i]
If (A == 0)
    A = W[i]
Else
    Y[i] = 0
Z[i] = A * X[i]
```



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

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)
```

```
0:  ldf                 Y(r1), f2
1:  fsbne               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)
```

Using Predication

ISA Support for Predication

- Intel Itanium: 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

```
0:  ldf          Y(r1), f2
1:  fsbne        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 Performance

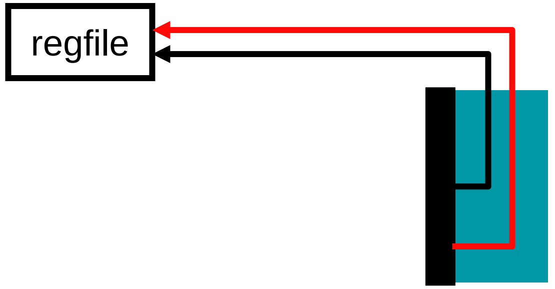
- Cost/benefit analysis
  - Benefit: Predication avoids branches
    - Thus avoiding mis-predication
    - Also reduces pressure on predicator table (fewer branches to track)
  - Cost: extra instructions (fetched, but not actually executed)
- As branch predictor are highly accurate …
  - Might not help:
    - 5-stage pipeline, two instruction on each path of if-then-else
    - No performance gain, likely slower if branch predictable
  - But can help
    - Deeper pipelines, hard-to-predict branches, and few added insn
- Thus, predication is useful, but not a panacea

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
  - **Option #4:** Pipeline the cache (“double pump”)
    - Start cache access every half cycle
    - Difficult circuit techniques

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
  - But some ISAs have update or auto-incr/decr addressing modes
- 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 instructions, 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

Very Long Instruction Word (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
- **Compiler-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 (maybe) still necessary
  - Typically "slotted": 1st insn must be ALU, 2nd mem, etc.
    - + Further simplification

Very Long Instruction Word (VLIW)

VLIW: instructions that "encode" multiple operations.

The hardware executes the entire instruction "at once" on parallel function units in execute stage.

The "long instructions" encode the fact that the operations are <u>independent</u>. So, the hardware does not need to dynamically figure this out. This can save area and power and is now popular in embedded processors (e.g., Texas Instruments C6x series of DSPs).

```
Inst N:       ADD.D  F1,F1,F4          NOP              L.D  F7, 0(R1)    DSUBI R2,R2,#1
NOP
Inst N+1:     ADD.D  F1,F1,F7          MUL.D F4,F5,F6   NOP          DSUBI R2,R2,#1
BEQZ  R2, Loop
```

VLIW Advantages

- + Simpler instruction fetch
  - Fetch a bundle instructions per cycle
- + Simpler dependence check logic
  - Compiler guarantees all instructions in bundle independent
- + Simpler branch prediction
  - Restrict to one branch per bundle
- By default, doesn’t help bypasses or register file problems
- Compiler-visible clustering possible in VLIW
  - Each “lane” of VLIW has “local” registers (read/written by this lane)
  - A few “global” registers (R/W by any lane) are used to communicate between lanes

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 and signal processors
    - Not useful for general-purpose computation

- **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

VLIW Disadvantages

- - Code density
    - Lots of “no-ops” in bundles
- Not compatible across machines of different widths
    - “Not compatible” could also mean programs would execute incorrectly
    - Or, “not compatible” can mean programs would execute slowly
- VLIW doesn’t solve all problems
    - VLIW mainly targets dependence checking
        - Which isn’t the worst N^2 problem in multiple-issue
    - Doesn’t magically create ILP

EPIC

- **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 (three 41-bit insns + 4 dependence bits)

Multiple Issue Redux

- **Multiple issue**
  - Exploits insn level parallelism (ILP) beyond pipelining
  - Improve IPC, but perhaps at some clock & energy penalty
  - 4-6 way issue is about the peak issue width currently justifiable
- **Problem spots**
  - N^2 bypass & register file -> clustering
  - Fetch + branch prediction -> buffering, loop streaming, trace cache
  - N^2 dependency check -> VLIW/EPIC

ILP and Static Scheduling

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

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

- Loop unrolling, software pipelining
  - Take compilers class for more detail

Takeaway Questions

- What are pros and cons of superscalar architecture ?
- What is the biggest challenge in the superscalar architecture ? Why?
- How to increase the data cache bandwidth ?
- What is VLIW ?
- What are pros and cons of VLIW ?