CS2410: Computer Architecture

Instruction Level Parallelism

Sangyeun Cho

Computer Science Department
University of Pittsburgh

What is instruction level parallelism?

- Execute *independent instructions* in parallel
  - Provide more hardware function units (e.g., adders, cache ports)
  - Detect instructions that can be executed in parallel (in hardware or software)
  - Schedule instructions to multiple function units (in hardware or software)
- Goal is to improve instruction throughput

- How does it differ from general parallel processing?
- How does it differ from pipelining?
  - Ideal CPI of single pipeline is 1
  - W/ ILP we want CPI < 1 (or IPC > 1)

Key questions

- How do we find parallel instructions?
  - Static, compile-time vs. dynamic, run-time
  - Data dependence and control dependence place high bars

- What is the role of ISA for ILP packaging?
  - VLIW approach vs. superscalar approach
  - EPIC approach (e.g., Intel IA64)

- How can we exploit ILP at run time?
  - Minimal hardware support (w/ compiler support)
  - Dynamic OOO (out-of-order) execution support

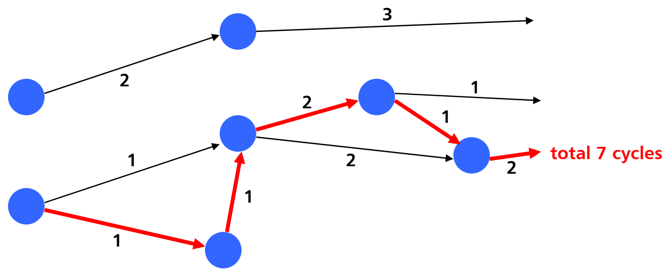
Data dependence

- Instructions consume values (operands) created by previous instructions

```
add r1, r2, r3
mul r4, r1, r5
```

- Given a sequence of instructions to execute, form a directed graph using producer-consumer relationships (not instruction order in the program)
  - Nodes are instructions
  - Edges represent dependences, possibly labeled with related information such as latency, etc.

Data dependence



- What is the minimum execution time, given unlimited resources?

- Other questions
  - Can we have a directed cycle?
  - What is the max. # of instructions that can be executed in parallel?
  - How do we map instructions to (limited) resources? In what order?

List scheduling (example impl.)

- Common compiler instruction scheduling algorithm

- Procedure
  - Build DPG (data precedence graph)
  - Assign priorities to nodes
  - Perform scheduling
    - Start from cycle 0, schedule "ready" instructions
    - When there are multiple ready instructions, choose the one w/ highest priority

List scheduling (example impl.)

```
cycle = 0
ready-list = root nodes in DPG
inflight-list = empty list
while ( ready-list or inflight-list not empty, and an issue slot is available )
    for op = (all nodes in ready-list in descending priority order)
        if (a functional unit exists for op to start at cycle)
            remove op from ready-list and add to inflight-list
            add op to schedule at time cycle
            if (op has an outgoing anti-edge)
                Add all targets of op's anti-edges that are ready to ready-list
            endif
        endif
    endfor
    cycle = cycle + 1
    for op = (all nodes in inflight-list)
        if (op finishes at time cycle)
            remove op from inflight-list
            check nodes waiting for op in DPG and add to ready-list
                if all operands available
        endif
    endfor
endwhile
```

(Cooper et al. '98)

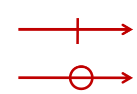
Name dependence

- Data dependence is "true" dependence
  - The consumer instruction can't be scheduled before the producer one
  - "Read-after-write"
- Dependences may be caused by the name of the storage used in the instructions, not by the producer-consumer relationship
  - These are "false" dependences

```
add r1, r2, r3
mul r2, r4, r5
```

```
add r2, r3, r4
…
mul r2, r5, r6
```

- Anti dependence ("write-after-read")
- Output dependence ("write-after-write")



Name dependence

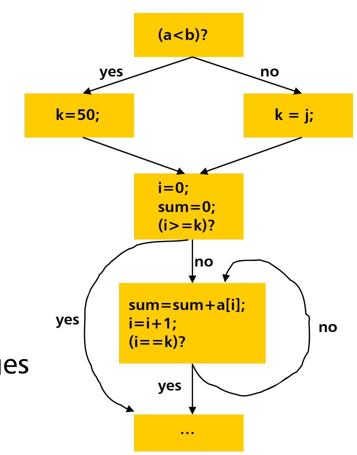
- Name dependences may be removed if we have plenty of storage (*i.e.*, many registers)
- In fact, some compilers first assume that there are unlimited registers
  - You can dump GCC internal representations (before register allocation) to confirm this
  - Compiler maps such "virtual" registers to "architected" registers
  - Compiler may generate code to store temporary values to memory when there are not enough registers
- Hardware can do the opposite – mapping architected registers ("virtual") to some physical register – to remove name dependences ⇒ register renaming
- Can we rename memory?

Control dependence

- It determines (limits) the ordering of *an instruction i* with respect to *a branch instruction* so that the instruction i is executed in correct program order and only when it should be
- Why are control dependences barriers for extracting more parallelism and performance?
  - Pipelined processor
  - Compiler scheduling

Control flow graph (CFG)

```
if (a<b)
   k=50;
else
   k = j;
for (sum=0,i=0;i < k; i++) {
   sum+=a[i];
}
```



- Nodes: basic blocks
- Edges: possible control changes
- How can you construct CFG?

Static vs. dynamic scheduling

- Static scheduling
  - Schedule instructions at compiler time to get the best execution time
- Dynamic scheduling
  - Hardware changes instruction execution sequence in order to minimize execution time
  - Dependences must be honored
  - For the best result, false dependences must be removed
  - For the best result, control dependences must be tackled
- Implementing precise exception is important

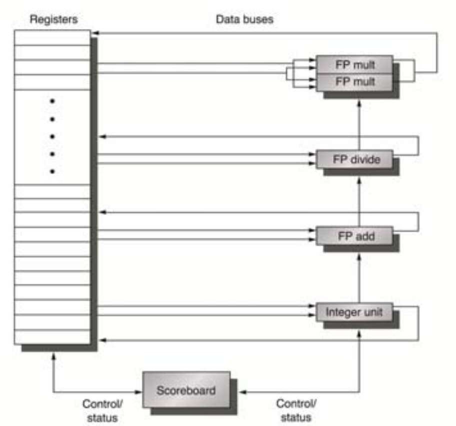
Dynamic scheduling

- Components
  - Check for dependences ⇒ "do we have ready instructions?"
  - Select ready instructions and map them to multiple function units
  - The procedure is similar to find parallel instructions from DPG
- Instruction window
  - When we look for parallel instructions, we want to consider many instructions (in "instruction window") for the best result
  - Branches hinder forming a large, accurate window
- We will examine two hardware algorithms: scoreboarding and Tomasulo's algorithm

CDC6600 scoreboard

- A dynamic scheduling method
  - A centralized control structure
  - Keeps track of each instruction's progress
- Instructions are allowed to proceed when resources are available and dependences are satisfied
- Out-of-order execution/completion of instructions

Basic structure



Tackling hazards

- Out-of-order execution of instructions may cause WAR and WAW hazards
  - They weren't interesting in in-order pipeline we examined ⇐ register read or write step in an earlier instruction is always before the write step of a later instruction
- Strategy
  - WAR
    - Stall write-back until all previous instructions read operands
  - WAW
    - Do not issue an instruction if there is another instruction that intends to write to the same register

Scoreboard control

- Issue (ID1)
  - Check for resource availability
  - Check for WAW
- Read operands (ID2)
  - Check for RAW (true dependency)
    - If no pending instructions will write to the same register, operands are fetched from the register file
    - Otherwise stall until operands are ready
  - Operands always come from register file ⇒ no forwarding
- Execution (EX)
  - FU starts execution on operands
  - When result is ready, FU notifies the scoreboard
- Write result (WB)
  - Check for WAR
    - Stall the completing instruction if there is dependency

Scoreboard data structures

- Instruction status: which of the 4 steps the instruction is in
- FU status: indicates the state of FU; 9 fields
  - Busy: is the FU busy?
  - Op: operation to perform (e.g., add or sub)
  - Fi: destination register
  - Fj, Fk: source registers
  - Qj, Qk: FU producing Fj and Fk
  - Rj, Rk: flags indicating if the associated operand has been read
- Register result status: which FU will generate a value to update this register?

Scoreboard example

	Instruction status			
Instruction	Issue	Read operands	Execution complete	Write result
L.D F6,34(R2)	√	√	√	√
L.D F2,45(R3)	√	√	√	
MUL.D F0,F2,F4	√			
SUB.D F8,F6,F2	√			
DIV.D F10,F0,F6	√			
ADD.D F6,F8,F2				

Functional unit status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	R3				No	
Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	Integer			Add	Divide			

Scoreboard example

	Instruction status			
Instruction	Issue	Read operands	Execution complete	Write result
L.D F6,34(R2)	√	√	√	√
L.D F2,45(R3)	√	√	√	√
MUL.D F0,F2,F4	√	√	√	
SUB.D F8,F6,F2	√	√	√	√
DIV.D F10,F0,F6	√			
ADD.D F6,F8,F2	√	√	√	

Functional unit status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			No	No
Mult2	No								
Add	Yes	Add	F6	F8	F2			No	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult 1			Add		Divide			

Scoreboard example

Instruction	Issue	Read operands	Execution complete	Write result
	Instruction status			
L.D F6,34(R2)	√	√	√	√
L.D F2,45(R3)	√	√	√	√
MUL.D F0,F2,F4	√	√	√	√
SUB.D F8,F6,F2	√	√	√	√
DIV.D F10,F0,F6	√	√	√	
ADD.D F6,F8,F2	√	√	√	√

Functional unit status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	No								
Mult2	No								
Add	No								
Divide	Yes	Div	F10	F0	F6			No	No

Register result status

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU						Divide			

Scoreboard limitations

- Small # of instructions available for parallel execution
  - Basic block in CDC6600
- Scoreboard size and organization (i.e., complexity)
  - ~instruction window
  - Centralized structure, not scalable
- Name dependences
  - There are WAR and WAW stalls

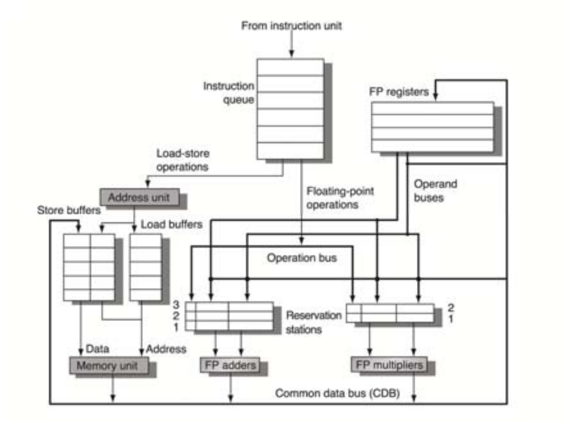
Tomasulo’s algorithm

- A dynamic scheduling algorithm
- Invented for IBM 360/91
- Motivation
  - High-performance FP without special compiler
  - Only 4 FP (architected) registers
  - Long memory and FP latencies

Tomasulo’s algorithm

- Register renaming
  - To overcome WAR and WAW (name dependences)
  - Provided by *reservation stations*
- Reservation stations (RS)
  - Operands are fetched in RS as they become ready (no restriction on their order)
  - Hazard detection and execution control are distributed
  - Results are passed directly to FUs from RS through *common data bus* (CDB)

Tomasulo's algorithm: hardware



Tomasulo's algorithm steps

- Issue
  - If there is an empty RS, get the next instruction from *instruction queue*
  - Fetch operands from register file or mark their producer FUs
- Execute
  - If operands become ready, the instruction can be executed in the FU
  - No unresolved branch allowed (this condition will be relaxed later)
- Write result
  - When result is obtained, write it on CDB
  - Stores write their data to memory
- Tagging data and objects
  - To recognize data of interest, each RS and load buffer is named
  - When a result is generated, it is launched on CDB with its tag, after the name of RS
  - Other objects snoop on CDB and catch the result if it is what they've waited for

Tomasulo's alg.: data structures

- Each RS has
  - Op: operation to perform
  - Qj, Qk: RS's to produce corresponding source operand; "zero" indicates that source operand is available in Vj or Vk
  - Vj, Vk: actual value of source operands
  - A: information for memory address calculation
  - Busy: whether RS and FU are busy or not
- Each register has
  - Qi: name of RS to produce the value to be written to this register

Tomasulo's algorithm: example

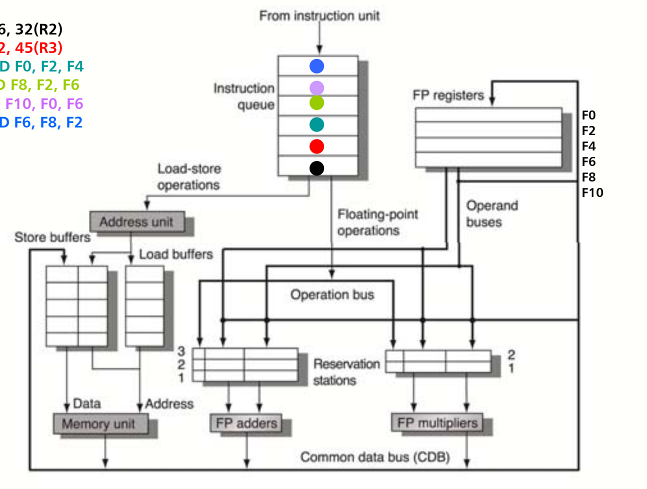
	Instruction status		
Instruction	Issue	Execute	Write Result
L.D F6,34(R2)	√	√	√
L.D F2,45(R3)	√	√	
MUL.D F0,F2,F4	√		
SUB.D F8,F2,F6	√		
DIV.D F10,F0,F6	√		
ADD.D F6,F8,F2	√		

			Reservation stations				
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	yes	Load					45 + Regs[R3]
Add1	yes	SUB		Mem[34 + Regs[R2]]	Load2		
Add2	yes	ADD			Add1	Load2	
Add3	no						
Mult1	yes	MUL		Regs[F4]	Load2		
Mult2	yes	DIV		Mem[34 + Regs[R2]]	Mult1		

				Register status					
Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

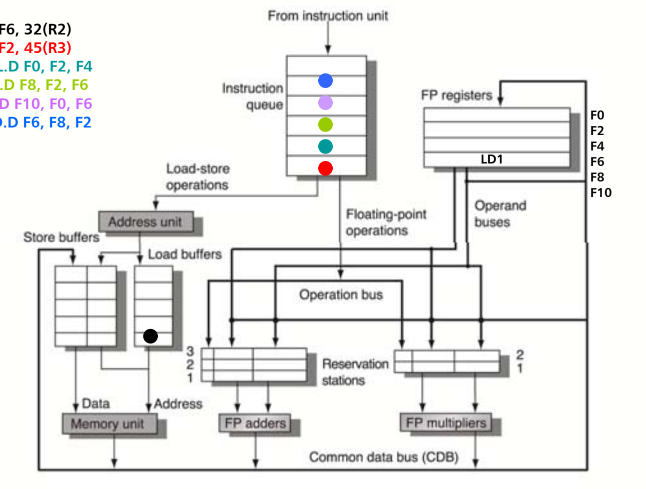
Tomasulo's algorithm: example

L.D F6, 32(R2)
L.D F2, 45(R3)
MUL.D F0, F2, F4
SUB.D F8, F2, F6
DIV.D F10, F0, F6
ADD.D F6, F8, F2



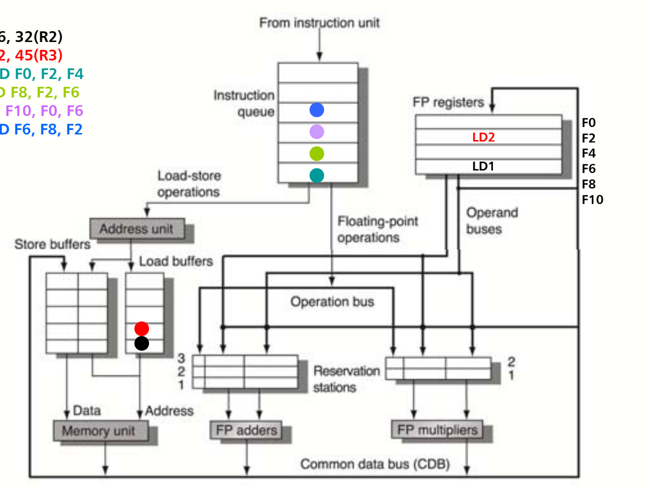
Tomasulo's algorithm: example

L.D F6, 32(R2)
L.D F2, 45(R3)
MUL.D F0, F2, F4
SUB.D F8, F2, F6
DIV.D F10, F0, F6
ADD.D F6, F8, F2



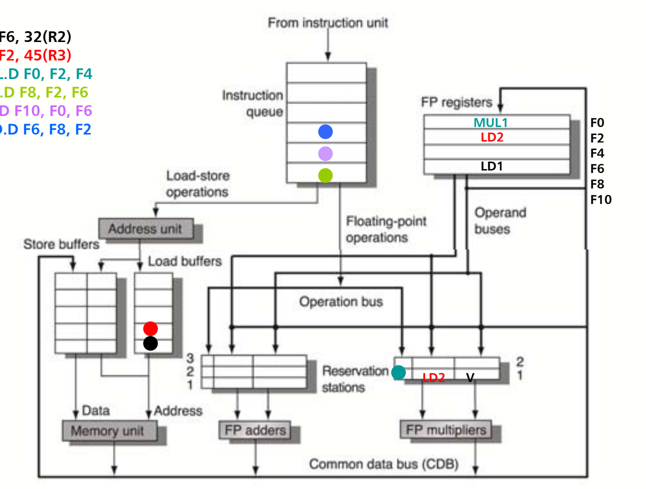
Tomasulo's algorithm: example

L.D F6, 32(R2)
L.D F2, 45(R3)
MUL.D F0, F2, F4
SUB.D F8, F2, F6
DIV.D F10, F0, F6
ADD.D F6, F8, F2



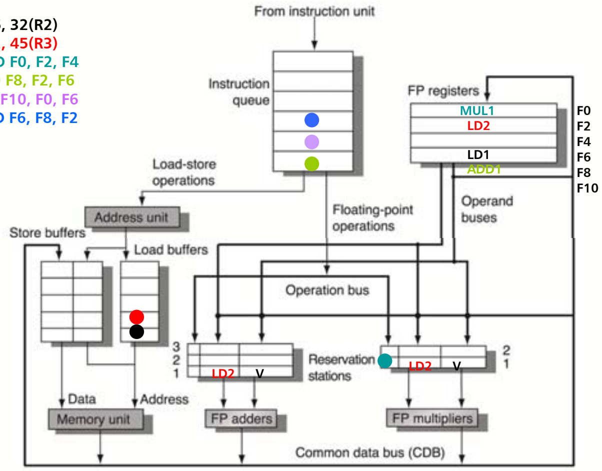
Tomasulo's algorithm: example

L.D F6, 32(R2)
L.D F2, 45(R3)
MUL.D F0, F2, F4
SUB.D F8, F2, F6
DIV.D F10, F0, F6
ADD.D F6, F8, F2



Tomasulo's algorithm: example

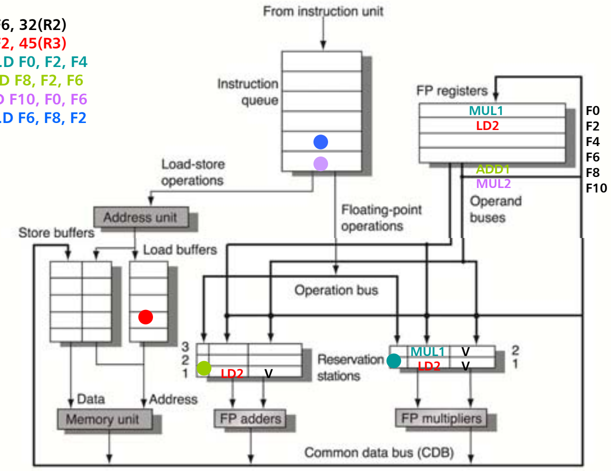
L.D F6, 32(R2)
L.D F2, 45(R3)
MUL.D F0, F2, F4
SUB.D F8, F2, F6
DIV.D F10, F0, F6
ADD.D F6, F8, F2



LD1 value broadcast

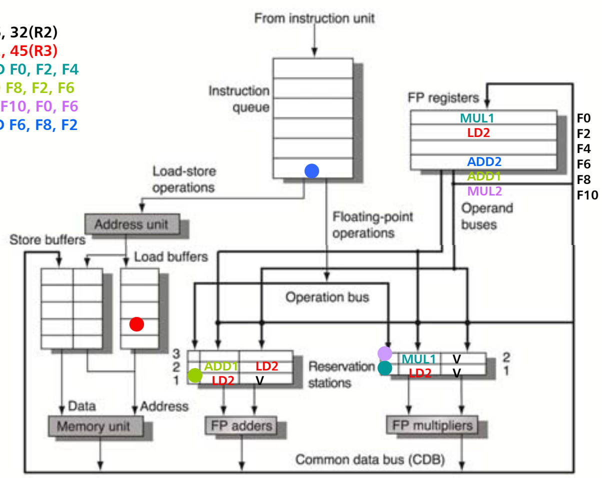
Tomasulo's algorithm: example

L.D F6, 32(R2)
L.D F2, 45(R3)
MUL.D F0, F2, F4
SUB.D F8, F2, F6
DIV.D F10, F0, F6
ADD.D F6, F8, F2



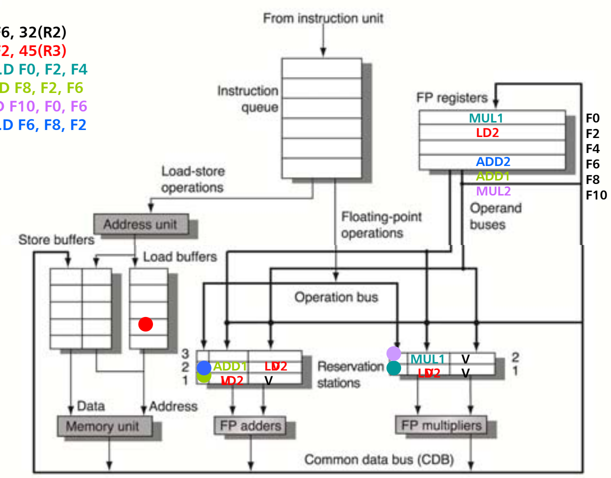
Tomasulo's algorithm: example

L.D F6, 32(R2)
L.D F2, 45(R3)
MUL.D F0, F2, F4
SUB.D F8, F2, F6
DIV.D F10, F0, F6
ADD.D F6, F8, F2



Tomasulo's algorithm: example

L.D F6, 32(R2)
L.D F2, 45(R3)
MUL.D F0, F2, F4
SUB.D F8, F2, F6
DIV.D F10, F0, F6
ADD.D F6, F8, F2



LD2 value broadcast

Tomasulo's algorithm: summary

- Register renaming
  - Automatic removal of WAR and WAW hazards
- Common data bus (CDB)
  - Broadcasting result (i.e., forwarding)
  - Need additional CDB to commit more than one instructions simultaneously
- Instructions are issued when an RS is free, not FU is free
- W/ dynamic scheduling
  - Instructions are executed based on value production & consumption rather than the sequence recorded in the program

Across branches

- A loop example (assume no branch prediction)

```
for (i=0; i< 1000; i++) {
  A[i] = B[i] * C[i];
  D[i] = E[i] / F[i];
}
```

```
for (j=1; j< 1000; j++) {
  A[j] = A[j – 1] / C[j];
}
```

```
for (k=0; k< 1000; k++) {
  Y = Y + A[k] / F[k];
}
```


Loop unrolling

```
for (i=0; i< 1000; i++) {
  A[i] = B[i] * C[i];
  D[i] = E[i] / F[i];
}
```

```
for (i=0; i< 1000; i+=4) {
  A[i] = B[i] * C[i];
  A[i+1] = B[i+1] * C[i+1];
  A[i+2] = B[i+2] * C[i+2];
  A[i+3] = B[i+3] * C[i+3];
  D[i] = E[i] / F[i];
  D[i+1] = E[i+1] / F[i+1];
  D[i+2] = E[i+2] / F[i+2];
  D[i+3] = E[i+3] / F[i+3];
}
```


Impact of branches

- We want to find more "ready" instructions for parallel execution
- 15~20% of all instructions executed are branches
  - Difficult to fetch instructions w/o stalls
  - Branch penalty (~# of issue width × branch resolution latency) becomes larger
- Uninterrupted instruction fetching
  - We need to know what to fetch (from where) every cycle
  - Branches should be predicted

Branch prediction

- What?
  - Taken or not taken (direction)
  - Target (where to jump to if taken)
- When?
  - When I fetch from the current PC
- (Boils down to get "next PC" given "current PC")
- How?
  - This is the topic of several slides
  - Let's assume a processor with a simple single-issue pipeline for presentation's sake

What to predict 1: T/NT

- Let's first focus on predicting taken (T) or not taken (NT)
- Static prediction
  - Associate each branch with a hint
    - Always taken
    - Always not taken
    - (Don't know)
  - Forward not taken, backward taken
  - Compiler hint
- Dynamic prediction
  - Simple 1-bit predictor
    - Remembers the last behavior
  - 2-bit predictor
    - Bias added
  - Combined
    - Choose between two predictors

1-bit predictor

- Remember the last behavior

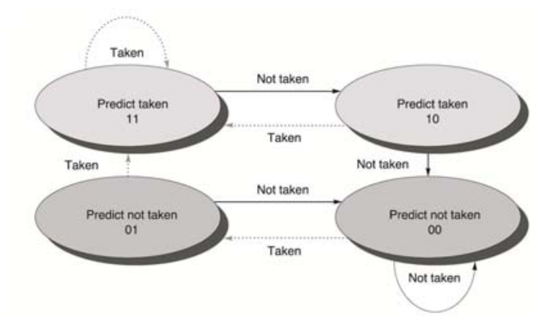
```
for (i=0; i< 100; i++) {
  A[i] = B[i] * C[i];
  D[i] = E[i] / F[i];
}
```

- How many hits and misses? Misprediction rate?

```
for (j=0; j<10; j++) {
  for (i=0; i< 100; i++) {
    A[i] = B[i] * C[i];
    D[i] = E[i] / F[i];
  }
}
```

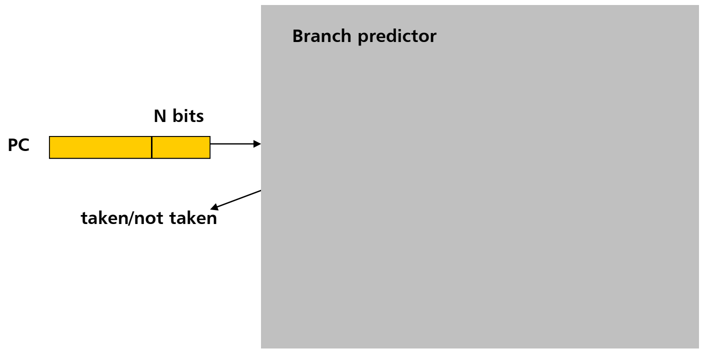

2-bit predictor

- Requires two consecutive mispredictions to flip direction

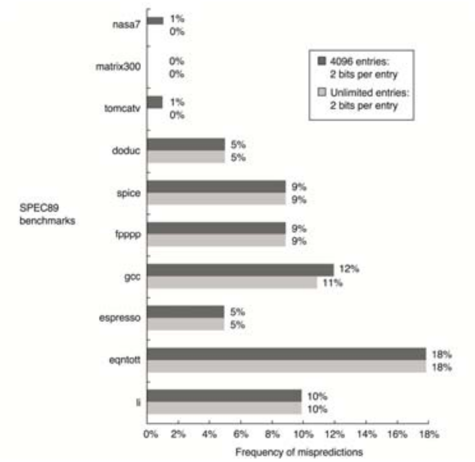


Branch prediction buffer

- Tag-less table to keep 2^N 2-bit counters, indexed with current PC



2-bit predictor performance



Correlating predictor

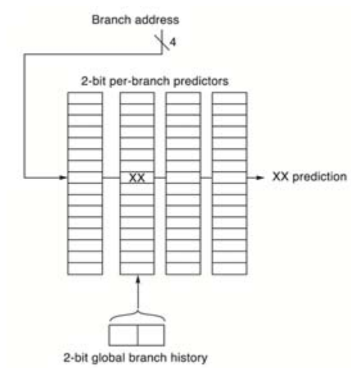
- Behavior of a branch can be correlated with other (previous) branches

```
if (aa==2)
  aa = 0;
if (bb==2)
  bb = 0;
if (aa!=bb) {
  …
}
```

- Branch history
  - N-bit vector keeping the last N branch outcomes (shift register)
  - 11001100 = TTNNTTNN (T being the oldest)
- Also called *global predictor*

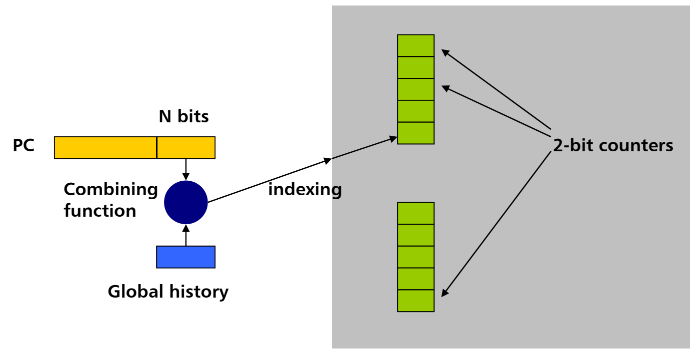
(m,n) predictor

- Consider last m branches
- Choose from 2^m BPBs, each has n-bit predictors

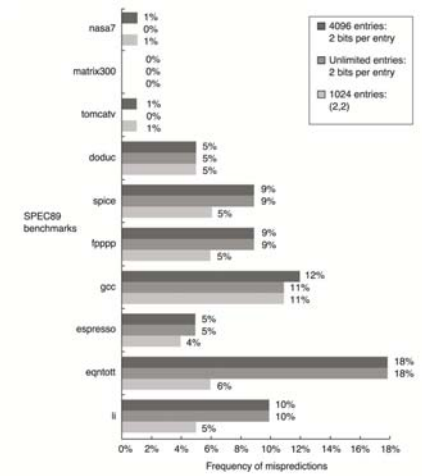


Combining index and history

- Form a single index from PC and GH



(2,2) predictor performance

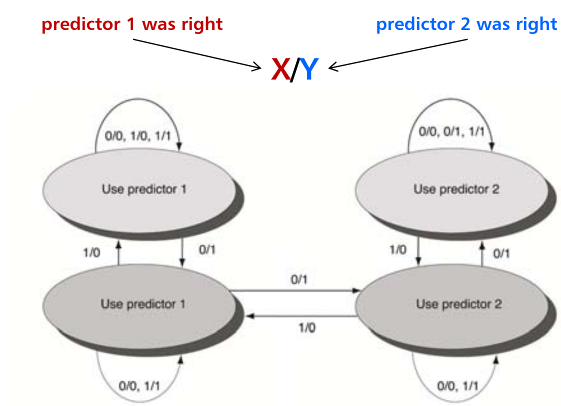


Combined predictor

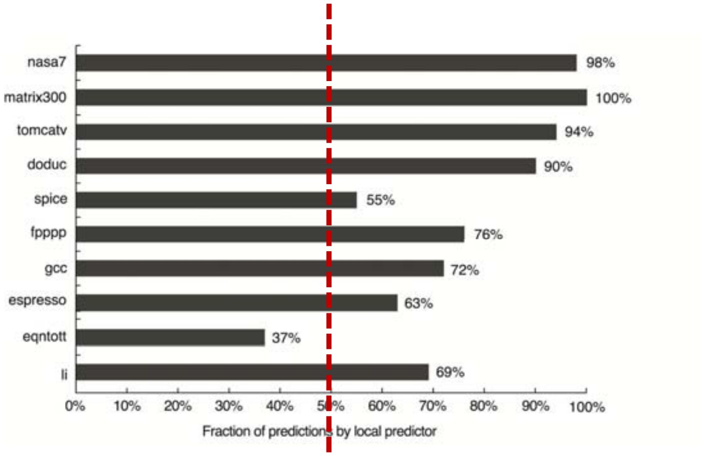
- Choose between local and global predictors
- Selector (again a predictor)

- Alpha 21264 example
  - 4K-entry global predictor (2-bit counters)
    - Indexed by 12-bit global history
  - Hierarchical local predictor
    - 1K 10-bit pattern table
    - 1K-entry 3-bit counters
  - Tournament predictor
    - 4K-entry 2-bit counters

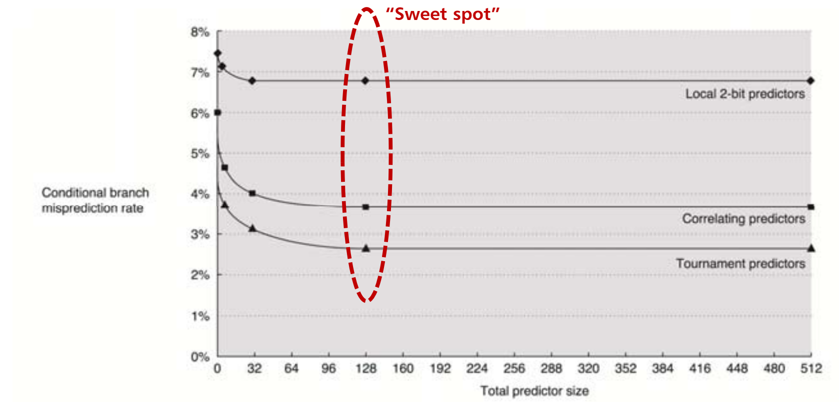
Selector design



Local vs. global?



Combined predictor performance



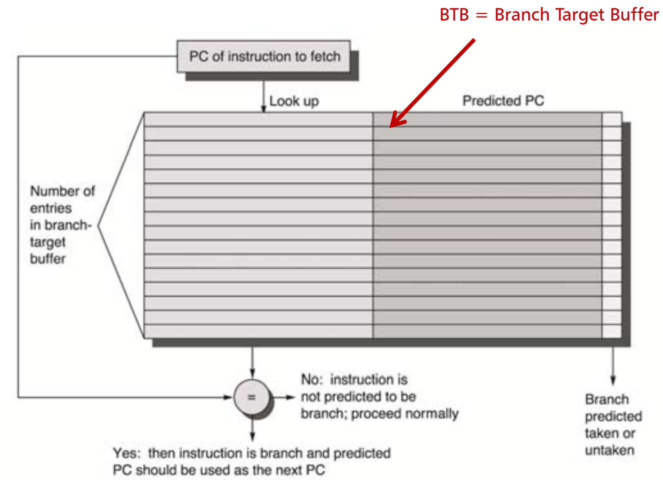
What to predict 2 – target

- Remember – the goal of branch prediction is to determine the next PC (target of fetching) every cycle
- Requirements
  - When fetching a branch, we need to predict (simultaneously with fetching) if it's going to be taken or not ⇐ we talked about this
  - At the same time, we need to determine the target of the branch if the branch is predicted taken ⇒ we are going to talk about this

Target prediction

- It's much more difficult to "predict" target
  - Taken/Not taken – just two cases
  - A 32-bit target has 2^{32} possibilities!
- But taken target remains the same!
  - Just remember the last target then…

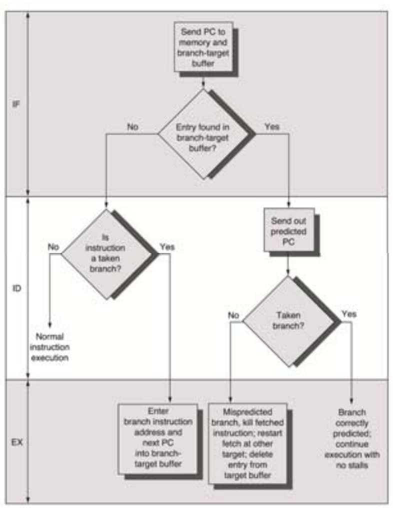
Target prediction w/ BTB



Target prediction w/ BTB

- Use "PC" to look up – why PC?
  - "Match" means it's a branch for sure
  - All-bit matching needed; why?
- If *match* and *Predicted Taken*, use the stored target for the next PC
- When no match and it's a branch (detected later)
  - Use some other prediction
  - Assume it's not taken
- After processing a branch, update BTB with correct information

Branch prediction & pipelining



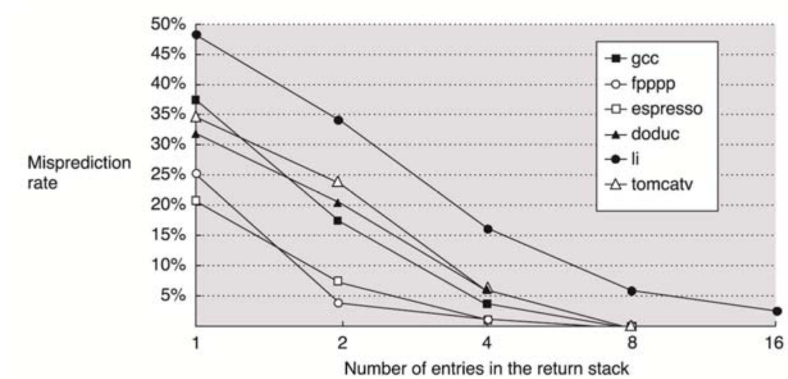
BTB performance

- Two bad cases
  - A branch is not found in BTB
  - Predicted wrongly
- Example: prediction accuracy is 90%, hit rate in the buffer is 90%, 2-cycle penalty, 60% of branches taken
  - Prob. (branch in buffer, mispredicted) = 90% × 10% = 0.09
  - Prob. (branch not in buffer, taken) = 10% × 60% = 0.06
  - Branch penalty = (0.09 + 0.06) × 2 = 0.30 (cycles)

What about "indirect jumps"?

- Indirect jumps
  - Branch target is not unique
  - E.g., jr $31
- BTB has a single target PC entry – can't store multiple targets…
- With multiple targets stored, how do we choose the right target?
- Fortunately, most indirect jumps are for *function return*
- Return target can be predicted using a stack ⇒ Return Address Stack (RAS)
  - The basic idea is to keep storing all the return addresses in a *Last In First Out* manner

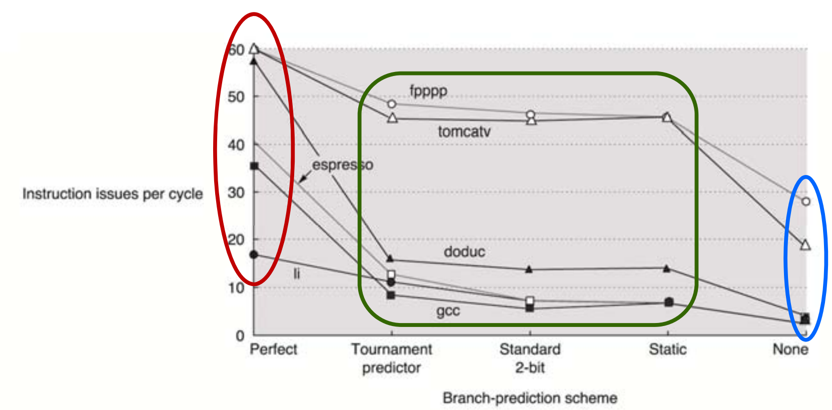
RAS performance



A few announcements

- HW #1 (graded) available to pick up
- We have the mid-term exam this Thursday; it will cover everything discussed until last week (including branch prediction)
- Regarding remaining homework assignments
  - I've reduced 10 assignments down to 8
  - HW #3 has been posted (due 10/28)

Performance of branch prediction

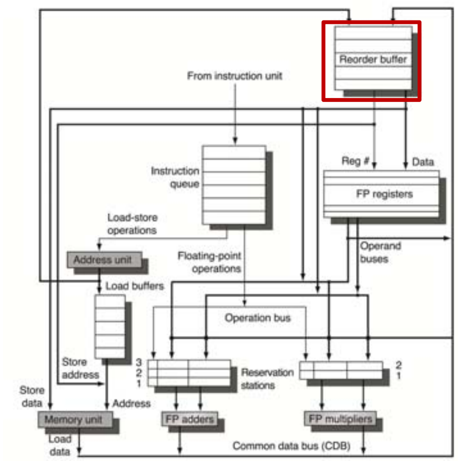


- On a hypothetical "64-issue" superscalar processor model with 2k instruction window

Speculative execution

- Execute instructions before their control dependences have been resolved
  - Execute instructions based on *speculation* (*i.e.*, branch prediction)
  - If speculation was right, we've done more useful work
  - If speculation was wrong, we need to cancel the effects that shouldn't have been caused
- Hardware speculation extends the idea of dynamic scheduling
- Issues
  - How and where to buffer speculative results?
  - How to cancel executed instructions if speculation was wrong?
  - How to implement precise exception?

Tomasulo's algorithm extended



Reorder buffer (ROB)

- In Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file
- With speculation, the register file is not updated until the instruction commits
- Thus, the ROB supplies operands in interval between completion of instruction execution and instruction commit
  - ROB is a source of operands for instructions like RS
  - ROB extends architected registers like RS

ROB entry

- Each entry in the ROB contains four fields:
- Instruction type
  - A branch (has no result to go to a register), a store (has a destination memory address), or a register operation
- Destination
  - Register number (for loads and ALU instructions) or memory address (for stores)
- Value
  - Value of instruction result until the instruction commits
- Ready
  - Indicates that instruction has completed execution and the value is ready

Speculative execution steps

- Issue
  - Get an instruction from instruction queue. Allocate RS and ROB entry.
  - Send operands from RF or ROB if available.
  - Record ROB entry name at RS (for tagging)
- Execute
  - If operand is not ready, monitor CDB. Execute when operands are ready and FU is idle.
- Write result
  - When result is ready, write it on CDB (tagged with ROB entry number).
  - ROB and awaiting RS's are updated.
- Commit
  - Normal commit: instruction reaching head of ROB with its result – update RF and remove it from ROB. Update memory if the instruction is store.
  - Branch: if the head instruction is a mispredicted branch, remove this branch and all the following instructions. Execution starts from the correct successor.

IPC > 1

- To achieve IPC > 1, all the pipeline stages should support higher bandwidth
  - High bandwidth i-cache and instruction fetch unit
  - High bandwidth decoding
  - Dynamic scheduling with multiple issue – multiple functional units
  - Multiple completion – multiple buses
  - Multiple commit – high bandwidth register file
- Smart implementation techniques are needed, not to increase clock cycle time
- Two approaches
  - Superscalar
  - VLIW

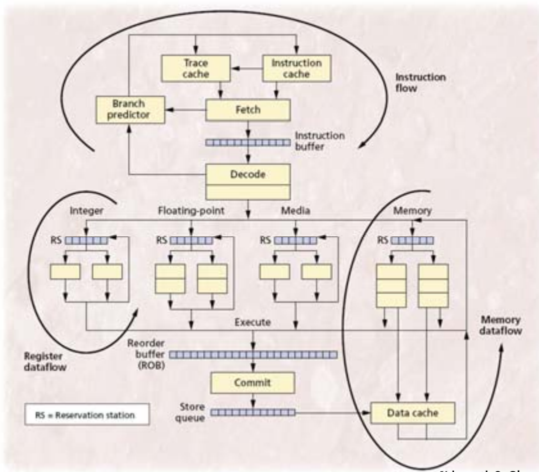
Superscalar processor

- A hardware-oriented design approach
  - Parallelism uncovered by hardware
  - Data dependence graph constructed at run time
  - Performance heavily dependent on speculation techniques and window size
- Binary compatibility easily maintained across processor generations
- Early superscalar processors executed 1 integer and 1 FP instructions (*e.g.*, Alpha 21164)
- Modern superscalar processors
  - Out-of-order execution/completion
  - Simultaneous multi-threading (SMT) built in
  - Deeply pipelined

VLIW: a compiler-oriented approach

- Parallelism detection done at compile time
  - Parallel instructions are packed into a long instruction word
    - Cheaper hardware impl. – no dependence checking between parallel instructions
  - Finding parallelism can be difficult
    - Frequent empty slots
- Extensive compiler techniques have been developed to find parallel operations across basic blocks
  - Trace scheduling, profile-driven compilation, …
  - Code size vs. performance
  - Code compatibility
- Recent examples
  - Transmeta's Crusoe
  - Intel's EPIC architecture (IA-64)
  - TI DSP processors

Three flows in a processor



[Lipasti & Shen 1997]

High-bandwidth i-cache

- I-cache should provide multiple instructions (say N instructions) per cycle from a given PC
- N instructions can span multiple cache blocks
  - Suppose the current PC points to the last instruction in a cache block
- Solutions
  - Multi-banked i-cache
    - e.g., IBM RS6000
  - Trace cache
    - e.g., Intel Pentium4

Instruction decode/issue

- Need to establish dependence relationship (i.e., data dependence graph) between multiple instruction in a cycle
  - With N instructions in considerations, $O(N^2)$ comparisons
  - What about previously buffered instructions?
- Need to look for multiple instructions when choosing a ready instruction to issue

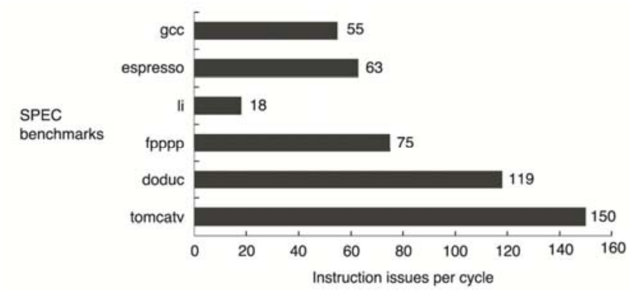
Multi-ported data cache

- There can be more than one memory accesses per cycle
  - Data cache must be multi-ported not to limit performance
- Multi-porting can be expensive
  - More area, power, and latency
- Example techniques
  - MIPS R10k: 2-port cache with interleaved multi-banking
  - Alpha 21164: 2-port cache with duplicated banking
  - Alpha 21264: 2-port cache with time-division multiplexing
  - Intel Itanium-2: 4-port cache with circuit-level multi-porting

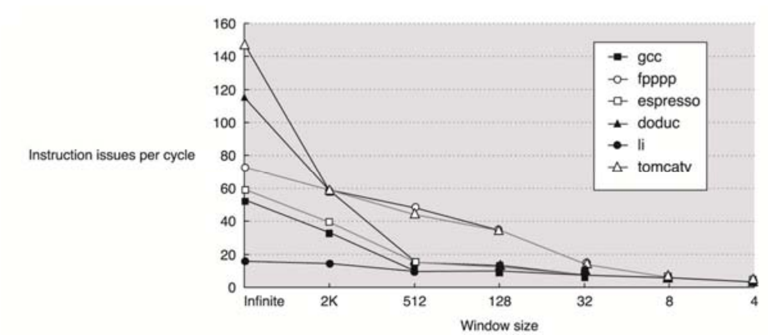
Limits of ILP

- What is the maximum (theoretical) ILP in programs?
- We need an ideal processor model
  - Register renaming with infinite physical registers
    - Only true dependences matter
  - Oracle branch prediction
    - No control dependences
  - Accurate memory address disambiguation
    - Memory accesses can be done as early as possible, out of order
  - Unlimited resources w/ an ideal 1-cycle latency (incl. memory access)

Optimistic ILP

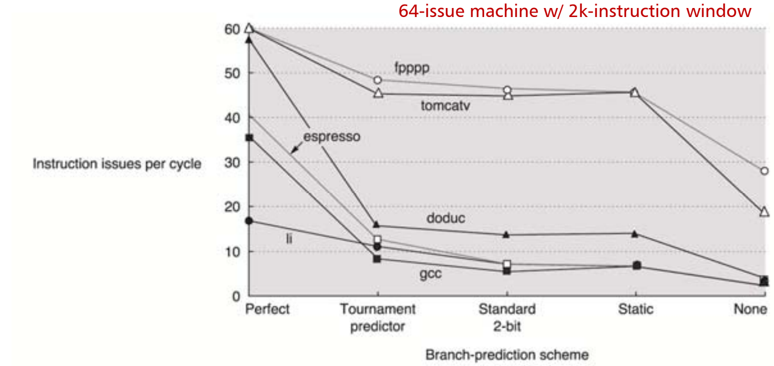


Window size

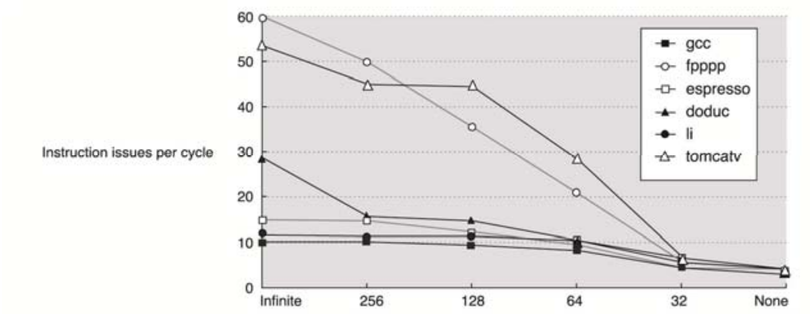


Impact of branch prediction

64-issue machine w/ 2k-instruction window

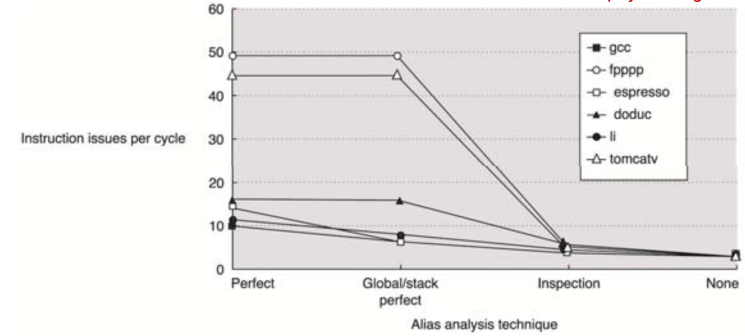


Fewer physical registers



Imperfect memory disambiguation

64-issue machine w/ 2k-instruction window 256/256 physical registers



ILP summary

- Fundamental barriers
  - Data dependence
  - Control dependence
- Instruction scheduling to extract parallelism
  - Static (before running your program)
  - Dynamic (when you run your program)
- Two dynamic scheduling hardware algorithms
  - CDC6600 scoreboarding
  - IBM 360/91 Tomasulo's algorithm
- Branch prediction for control-speculative execution
  - Local, global (correlated), combined
  - There are many other smart techniques, e.g., using neural network

- Today's superscalar processors mostly rely heavily on dynamic scheduling and other hardware techniques for higher performance; but they do benefit from sophisticated compilers

ILP summary

- Limits of ILP
  - Potential ILP (50?) vs. realizable ILP (2?)
- Limitations in hardware implementation
  - Data dependence among registers
    - Limited # of physical registers
  - Data dependence among memory references
    - Limited static/dynamic memory disambiguation
  - Control dependence
    - Sophisticated branch prediction, speculative execution, predicated execution, …
  - Scalability of key structures
    - Fetch unit, decode unit, execution pipelines, cache ports, …
- Hardware-based OOO vs. VLIW

- There is a diminishing return as we invest more resources to exploit as much ILP as possible $\Rightarrow$ turn to other forms of parallelism, e.g., thread-level parallelism
  - How do we achieve higher performance from an inherently single-threaded program?

Revisiting loop unrolling

```
for (i=0; i< 1000; i++) {
  A[i] = B[i] * C[i];
  D[i] = E[i] / F[i];
}
```

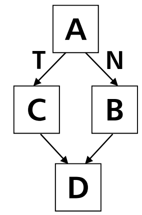
```
for (i=0; i< 1000; i+=4) {
  A[i] = B[i] * C[i];
  A[i+1] = B[i+1] * C[i+1];
  A[i+2] = B[i+2] * C[i+2];
  A[i+3] = B[i+3] * C[i+3];
  D[i] = E[i] / F[i];
  D[i+1] = E[i+1] / F[i+1];
  D[i+2] = E[i+2] / F[i+2];
  D[i+3] = E[i+3] / F[i+3];
}
```

- Positive effects
  - Less loop overhead
  - Better scheduling in the loop body
    - More parallel operations
  - Eliminate very small loops
    - More opportunities for code motion
- Problems
  - Code size increase
  - What if the loop count is not known at compile time?
  - What about a *while* loop?

Predicated execution

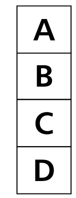
```
if (cond) {
  b = 0;
}
else {
  b = 1;
}
```

(normal branch code)




```
A  p1 = (cond)
   branch p1, TARGET
B  mov b, 1
   jmp JOIN
C  TARGET:
   mov b, 0
```

(predicated code)




```
A  p1 = (cond)
B  (!p1) mov  b, 1
C  (p1) mov  b, 0
```


Function inlining

- Replace a function call instance ("call foo()") with the actual function body ("foo()")
  - Similar to loop unrolling in a sense
- Similar benefits to loop unrolling
  - Remove function call overhead
    - Call/return (and possible branch mispredictions)
    - Argument/return value passing, stack allocation, and associated spill/reload operations
  - Larger block of instructions for scheduling
- Similar problems
  - Primarily code size increase

Trace scheduling

- Trace scheduling divides a procedure into a set of frequently executed *traces* (paths)
  - Make frequent traces run fast (common case)
  - Trace scheduling may make infrequent paths run slower (rare case)
- Three steps
  - Select a trace
    - Frequency information derived statically or from profile data
  - Schedule a trace
    - Aggressively schedule instructions as if there are no branches into and out of the trace
  - Insert fix-up code
    - Take care of mess

Trace scheduling

```
a=log(x);
if(b>0.01){
  c=a/b;
}else{
  c=0;
}
y=sin(c);
```

```
a=log(x);          fixit:
c=a/b;               c=0;
y=sin(c);            y=0; // sin(0)
if(b<=0.01)
  go to fixit;
```

Suppose profile says that b>0.01 90% of the time

Now we have larger basic block for our scheduling and optimizations

Price of fix-up code

- Assume the code for b>0.01 accounts for 80% of the time
- Optimized trace runs 15% faster
- Fix-up code may cause the remaining 20% of the time slower!
- Assume fix-up code is 30% slower

By Amdahl's Law:

Speedup = 1/(0.2+0.8*0.85)
= 1.176

17.6% performance improvement!

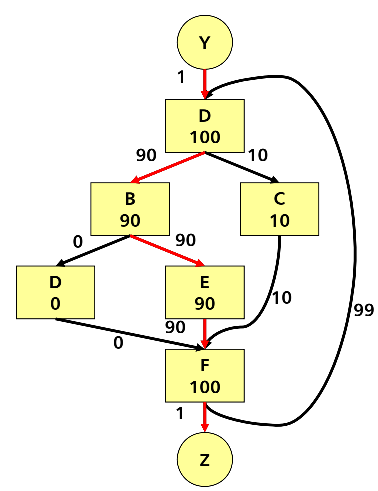
Speedup = 1/(0.2*1.3+0.8*0.85)
= 1.110

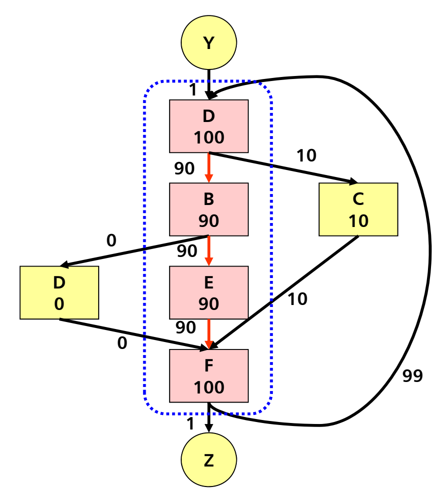
Over 1/3 of the benefit removed!

Superblock

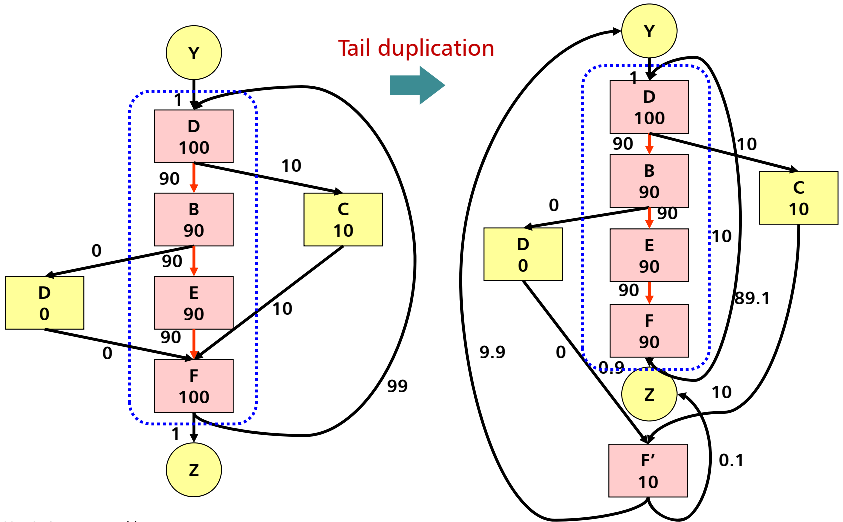
- Inserting fix-up code for traces can be quite complex, especially in the presence of many branch outlets and aggressive code motion
- A *superblock* is a trace without side entrances; control can only enter from the top, but it can leave at one or more exit points

Superblock formation

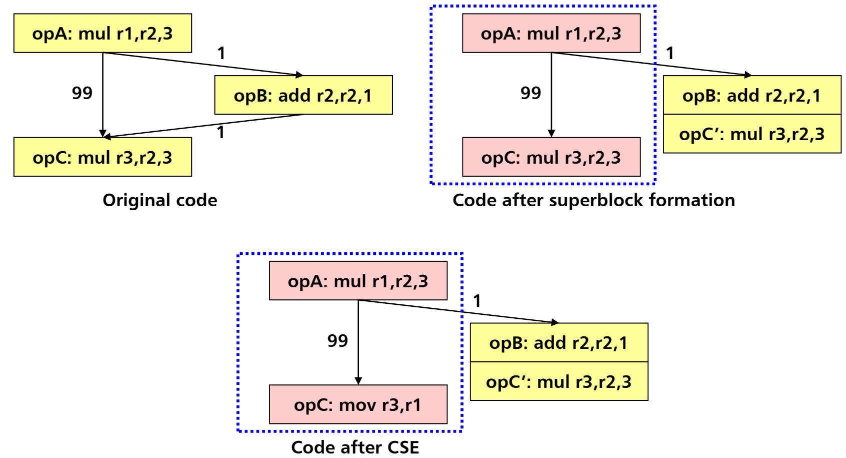




Superblock formation



CSE in Superblock

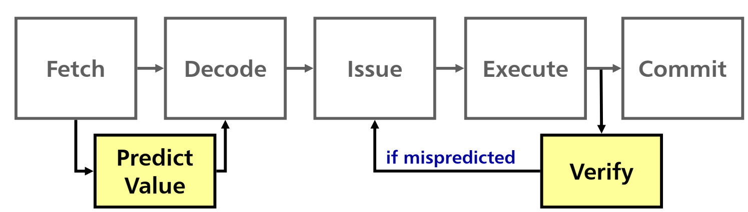


Value prediction

- Data dependence places fundamental limitation
  - You can't achieve a shorter latency than the maximum path length in the data precedence graph of a program
- What about predicting a value before computation (just like we predict the outcome of a branch)?
  - Branch prediction: binary (T or NT)
  - Value prediction: 2^{32} or 2^{64}
  - Is it possible to predict a value?
- With successful value prediction, you may be able to break the data dependence chains!

Value prediction

- Speculative prediction of register values
  - Values predicted during fetch and decode stages, forwarded to dependent instructions
  - Dependent instructions can be issued and executed immediately
  - Before committing instructions, we must verify the predictions; if wrong, we must restart instructions that used wrong values



[Lipasti & Shen 1996]

Classifying *speculative execution*