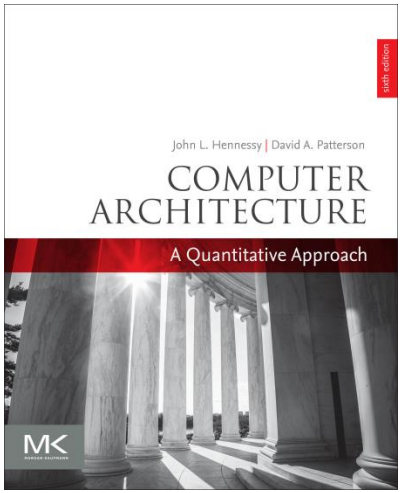
CESE 4085 Modern Computer Architecture

Lectures 5 & 6: Memory Hierarchy Design



Chapter 3

Instruction-Level Parallelism and Its Exploitation

Lecture Overview

Lecture 5:

- ILP – Dependences – Hazards
- Branch Prediction (lab topic)
- Dynamic Scheduling – Register Renaming
- Tomasulo's Algorithm

Combining topics of 6th and earlier editions of CA book

Lecture 6:

- Speculation – Reorder Buffer
- VLIW Processors
- Branch Target Buffer (BTB), Return Address Predictor
- Fallacies and Pitfalls

Introduction

Pipelining become universal technique in 1985

- Overlaps execution of instructions
- Exploits "**Instruction Level Parallelism**" (next slide)

Beyond this, there are two main approaches:

- Hardware-based dynamic approaches
  - Used in server and desktop processors
  - Not used as extensively in PMP processors
- Compiler-based static approaches
  - Not as successful outside of scientific applications

Instruction-Level Parallelism

Pipelining (partially) overlaps the execution of instructions that are independent

instruction 1	IF	ID	EX	MEM	WB		
instruction 2		IF	ID	EX	MEM	WB	
instruction 3			IF	ID	EX	MEM	WB

This type of parallelism is called *Instruction-Level Parallelism* (ILP)

Multiple-issue processors exploit even more ILP by fetching and executing multiple instructions in parallel

- Dynamic issue, superscalar processor: *hardware* tries to find and execute independent instructions in parallel
- Static issue, VLIW processor: *compiler* finds and groups independent instructions in one VLIW which is executed in parallel by the processor

ILP Processors

Dynamic approaches dominate the desktop and server markets:

- Pentium III and 4
- Athlon
- MIPS R10000/12000
- Sun ULTRASPARC III
- PowerPC 603, G3, and G4
- Alpha 21264

Static approaches are often seen in the embedded market (IA64/Itanium is an exception):

- TriMedia TM32

Multi-threading slides

There is another location in the story about multithreading.

This is in the new slides, not old slides.

Include??

Pipeline CPI

Pipeline CPI = Ideal pipeline CPI

+ Structural Hazard Stalls

+ Data Hazard Stalls + Control Hazard Stalls

(+ Memory Stall Cycles + Write Buffer Stalls)

Ideal pipeline CPI: measure of the maximum performance attainable by the implementation

Structural hazards: HW cannot support this combination of instructions (Un-pipelined FU, single memory pipeline for I & D)

Data hazards: Instruction depends on result of prior instruction still in the pipeline

Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

Parallelism within basic block is limited

typical size of basic block = 3-6 instructions

must optimize across basic blocks

Dependences versus Hazards

Dependences are properties of programs

Hazards are properties of a machine organization

If the pipelined processor had no forwarding hardware, any data dependence between consecutive instructions would result in a RAW hazard:

- Think of where writeback stage is !!

Data Dependence

Loop-Level Parallelism

- Unroll loop statically or dynamically
- Use SIMD (vector processors and GPUs)

Challenges:

- Data dependency
  - Instruction *j* is data dependent on instruction *i* if
    - Instruction *i* produces a result that may be used by instruction *j*
    - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i*

Dependent instructions cannot be executed simultaneously

Data Dependence

Dependencies are a property of programs

Pipeline organization determines if dependence is detected and if it causes a stall

Data dependence conveys:

- Possibility of a hazard
- Order in which results must be calculated
- Upper bound on exploitable instruction level parallelism

Dependencies that flow through memory locations are difficult to detect

Name Dependence

Two instructions use the same name but no flow of information

- Not a true data dependence, but is a problem when reordering instructions
- Antidependence: instruction j writes a register or memory location that instruction i reads
  - Initial ordering (i before j) must be preserved
- Output dependence: instruction i and instruction j write the same register or memory location
  - Ordering must be preserved

To resolve, use register renaming techniques (next lecture)

Other Factors

Data Hazards

- Read after write (RAW)
- Write after write (WAW)
- Write after read (WAR)

Control Dependence

- Ordering of instruction i with respect to a branch instruction
  - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
  - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

Hazards – more explanation

- structural: contention at shared resources
- data hazards:
  - RAW: most common; due to true data dependencies
  - WAW: due to output dependencies
    - N>1 operations write in the same register
  - WAR: due to antidependencies
    - instruction *j* (over)writes a register before it was read by a previous (earlier in program order) instruction *i*
    - cannot occur in static pipelines (why?)
  - forwarding ensures certain dependences do not result in hazards or stalls
  - dependencies that cannot be hidden result in stalls
- control hazards: branch prediction (next set of slides)

Start - Branch Predictors

Dynamic Branch Prediction

decrease control hazards stalls

Tomasulo's algorithm stops issuing instructions when a branch is encountered

on average 15 to 25% instructions are branches, hence the maximum ILP is 3 to 6

situation even worse for multiple-issue (superscalar) processors, because we need to provide an instruction stream of *n* instructions per cycle.

Idea: predict the outcome of branches based on their history and execute instructions speculatively

6 Branch Prediction Schemes

1-bit Branch Prediction Buffer

2-bit Branch Prediction Buffer

Correlating Branch Prediction Buffer

Tournament Predictors

Branch Target Buffer

Return Address Predictors

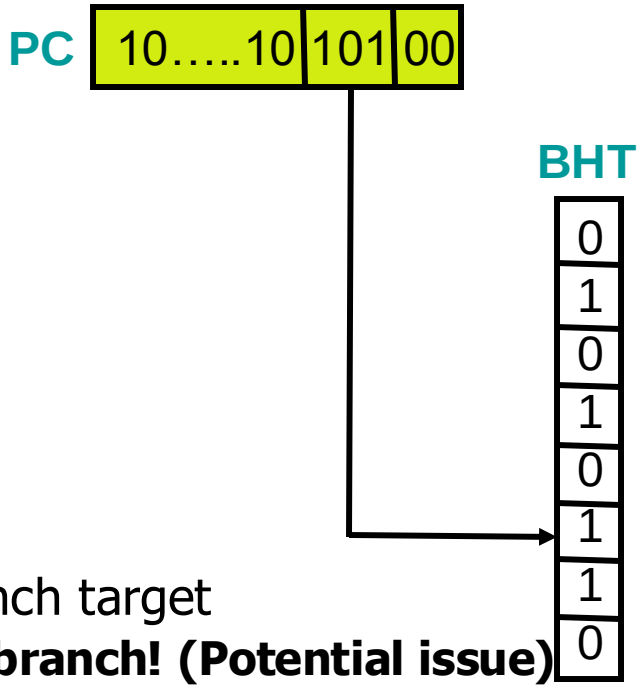
+ A way to get rid of those malicious branches

Before we continue!

Let's take a step back!
Q: How to predict a single branch?
A: Use a single bit to store the previous outcome and use this as prediction
Q: How to avoid "flip-flopping" of this bit?
A: Add more bits ☺
Q: How to call this approach?
A: (0,n) with n=# bits
Q: What does the zero mean?
A: It means "independent of other branches"
Q: What does (1,2) or (2,2) mean?
A: Branch predictor is correlated with preceding 1,2 branches, respectively
Q: How many "tables" are we going to maintain for (2,2) branch predictor?
A: what are possibilities? NT/NT, NT/T, T/NT, T/T – thus 4 tables!!
And, each table has 2-bit entries (per row)

1-bit Branch Prediction Buffer

- 1-bit *branch prediction buffer* or *branch history table*. Entry is 0 or 1 if the branch was not taken (then PC=PC+4) or taken (then PC=PC+x) the previous time.



- BHT is indexed by the lower bits of the branch target
  - **an entry might have been set by other branch! (Potential issue)**
- Does not help for 5-stage MIPS pipeline because target address is computed in same stage as branch condition

1-bit Branch Prediction Buffer

Code snippet from motion estimation (MPEG):

```
sad = 0;
for (i=0; i<16; i++)
   for (j=0; j<16; j++){
      if ((v = a[i][j]-b[i][j]) < 0) v = -v;
      sad += v;
   }
 //j=16        predict taken -> miss -> not taken
 //i>0 & j=0   predict not taken -> miss -> taken
 //i=16        predict taken -> miss -> not taken
```

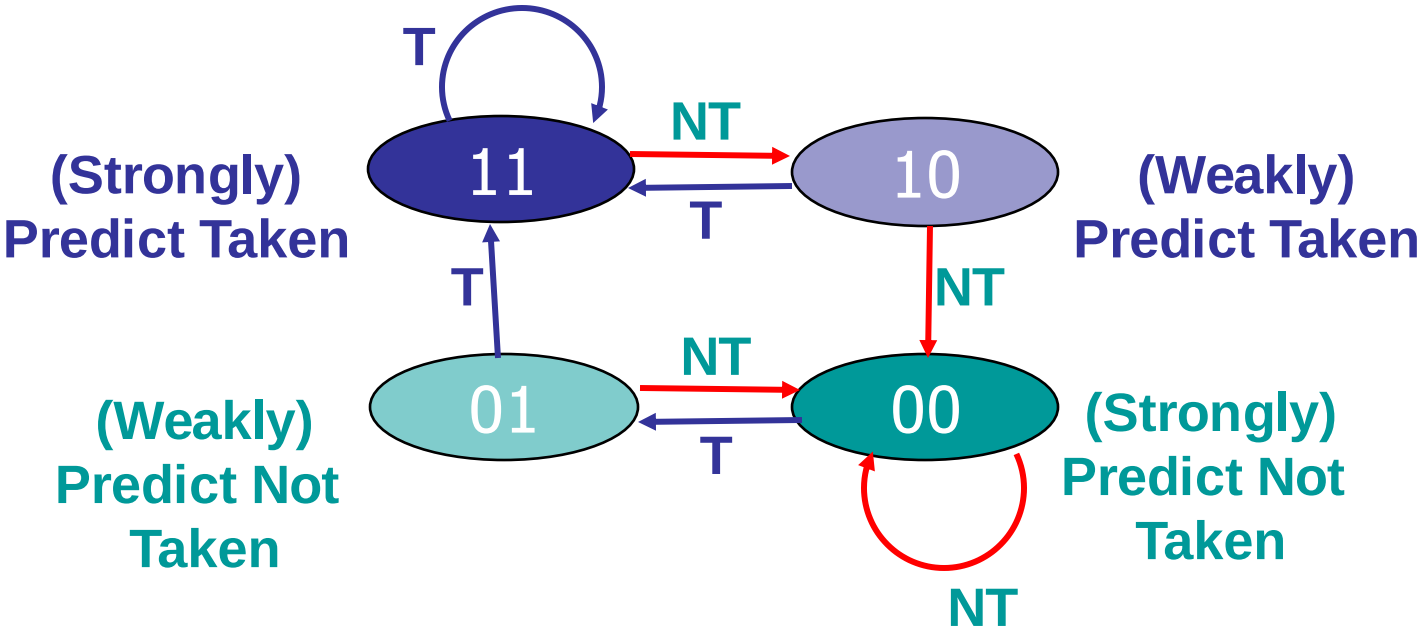
1-bit prediction: flips the BHT entry when mispredicted

Problem: in a nested loop, 1-bit BHT will cause 2 mispredictions:

- End of loop case, when it exits instead of looping as before
- At i>0, at first iteration of inner-loop the prediction is exit instead of looping
- Only 88% accuracy even if loop 94% of the time

2-bit Branch Prediction Buffer

Solution: 2-bit scheme where prediction is changed only if mispredicted twice:



Correlating Branches

Fragment from SPEC92 benchmark **eqntott**:

```
//R1=aa; R2=bb;
if (aa==2)          daddi R3,R1,#-2; R3=aa-2
   aa = 0;     b1:  bne   R3,R0,L1  ; NT if aa==2
if (bb==2)          dadd  R1,R0,R0
   bb = 0;     L1:  daddi R3,R2,#-2; R3=bb-2
if (aa!=bb){    b2: bne   R3,R0,b3  ; NT if bb==2
                    dadd  R2,R0,R0
                b3: beq   R1,R2,L3  ; T if aa==bb
```

If **b1** and **b2** are both not taken, then **b3** will be taken.

If both are taken, then **b3** will not be taken

Correlating Branch Predictors

Behavior of a branch not only depends on its own history (local history) but also on the behavior of previously executed branches (global history)

Suppose we look at the 2 previously executed branches. Then there are 4 possibilities:

- NT/NT (00)
- NT/T (01)
- T/NT (10)
- T/T (11)

For each of these 4 possibilities, we maintain a local history for this branch

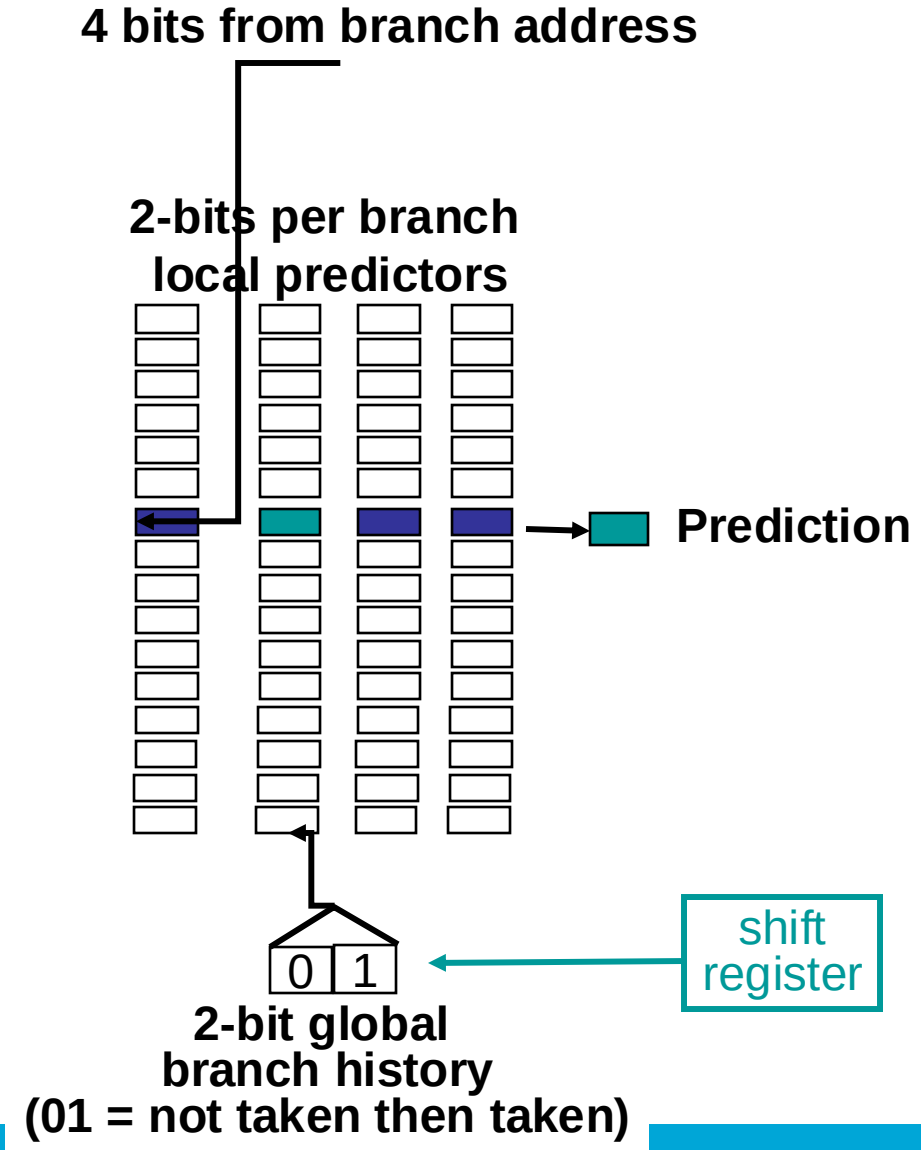
Correlating Branch Predictor

e.g., behavior of 2 recent branches selects between 4 predictions of next branch, updating just that prediction

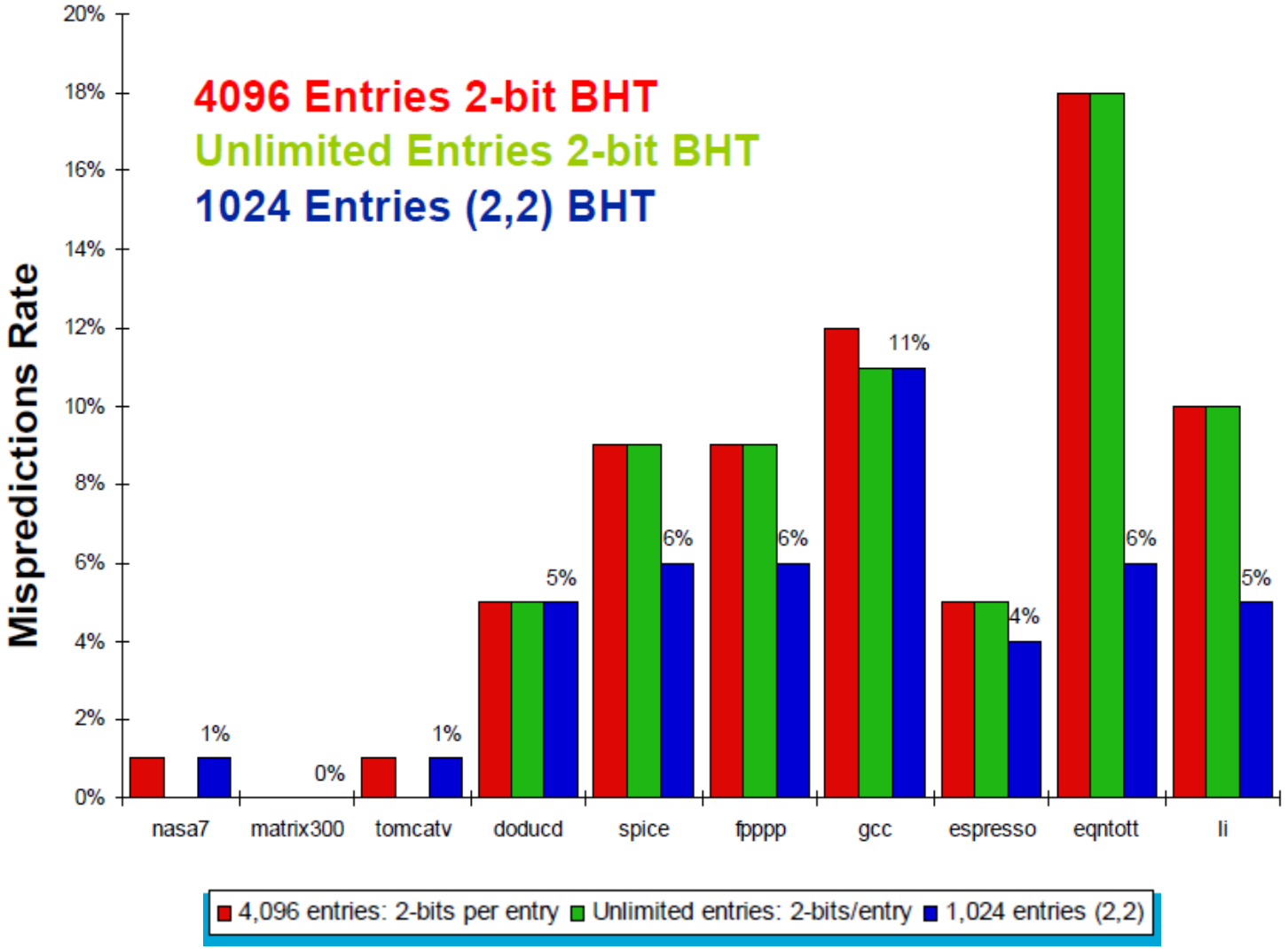
(2,2) predictor: 2-bit global, 2-bit local

- if two most recent branches were not taken (00), use first prediction
- 01: use second prediction
- 10: use third prediction
- 11: use fourth prediction

(*m*,*n*) predictor uses behavior of last *m* branches to choose from 2^m predictors, each of which is *n*-bit predictor



Accuracy of Different Branch Predictors



BHT Accuracy

Mispredict because either:

- Wrong guess for that branch
- Got branch history of wrong branch when indexing the table

4096 entry table: misprediction rates vary from 1% (nasa7, tomcatv) to 18% (eqntott), with spice at 9% and gcc at 12%

For SPEC92, 4096 about as good as infinite size table

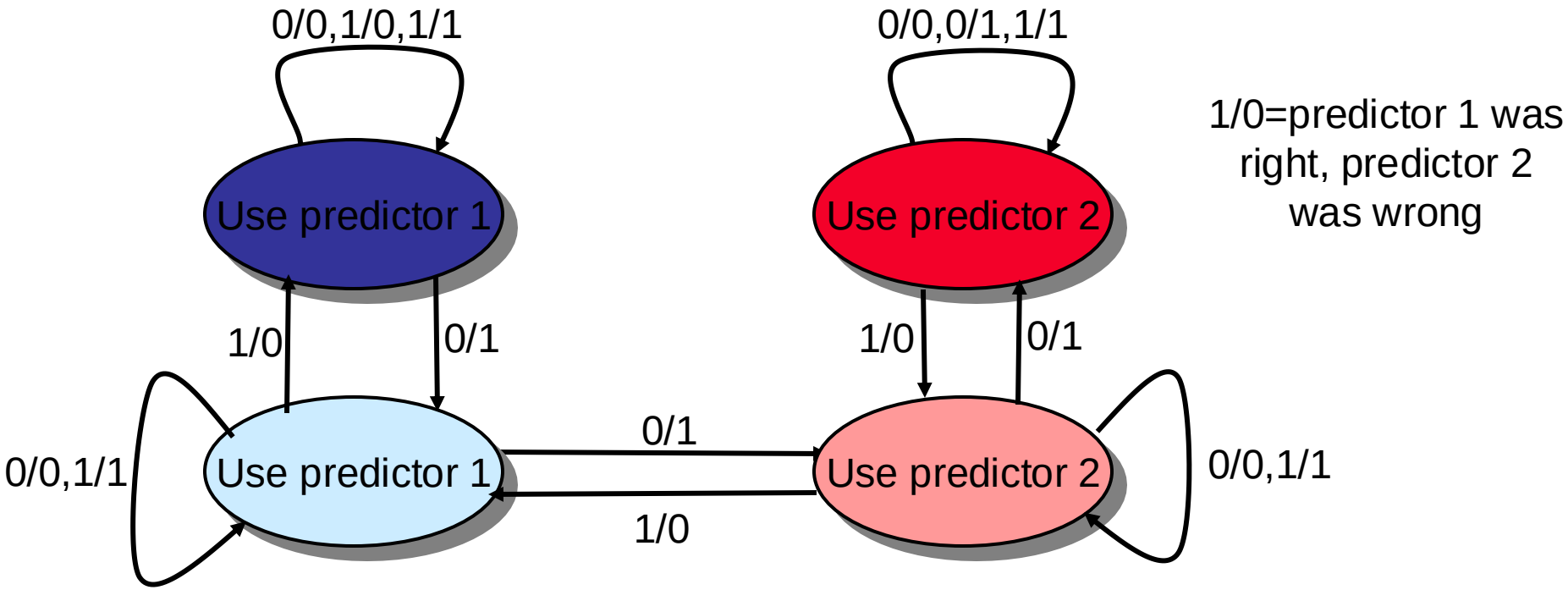
For some benchmarks, correlated branch predictor much better

Real programs + OS more like gcc

Tournament Predictor

Use multiple predictors, usually one based on local and one based on global information

Select between them depending on their accuracy



Alpha 21264 uses a very advanced tournament predictor

Dynamic Branch Prediction Summary

Prediction has become important part of (super)scalar execution

Branch History Table: 2 bits for loop accuracy

Correlation: Outcome of a branch depends on recently executed branches

- Either different branches
- Or different executions of same branch

Intermezzo – Side-story

Skip

The following set of slides is being skipped during the lecture, but is being kept here since it completes the story of branch prediction.

However, during this story, the Reorder Buffer (ROB) is being introduced and without the introduction of the Tomasulo algorithms, this is hard to follow.

Later on, after the Tomasulo algorithms has been introduced, this part of the story can be revisited.

The 6th edition discusses the BTB, RAP, speculation, ROB in a different order at a later stage.

Branch Target Buffer

Skip

Is NOT branch prediction!

Try to predict the **target address** of taken branches, unconditional branches, jumps – put this in a **buffer**

Q: Why is this interesting? What problem is being "solved"?

Hint 1/Q: related to I-cache or D-cache?

Hint 2/Q: what can we do when we "know" or can predict the target address?

A: We can now prefetch instructions (in the I-cache)!!

Return Address Predictor

Skip

Q: What does "return" mean in this case?

A: Return from procedure call

Easy to predict if always the same location (in code) is calling procedures!

Q: What if many different locations are calling the same procedure? How to predict in this scenario?

A: Instead of prediction, use **return stack** !

Push (return address = PC+4) when calling procedure & Pop (return address) when returning

Note: also works well for nested procedure calls!

Predicated Instructions

Skip

avoid branch prediction by turning branches into conditional or predicated instructions: if (x) then A = B op C else NOP

- If false, then neither store result nor cause exception
- Extended ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can annul any following instr.
- IA-64/Itanium: conditional execution of any instruction

```
if (R1==0) R2 = R3;          CMOVZ  R2,R3,R1

if (R1 < R2)                 SLT    R9,R1,R2
   R3 = R1;                  CMOVNZ R3,R1,R9
else                         CMOVZ  R3,R2,R9
   R3 = R2;
```

- Both CMOVNZ/CMOVZ can be issued early & simultaneously!

adds logic that might increase the processor cycle

limited use for complex control dependencies

Putting it all together: speculation

Skip

Dynamic branch prediction

- determines what instructions to fetch before a branch is resolved

Dynamic scheduling

- execute instructions out-of-order, when their data and control dependencies are resolved
- without branch prediction, only inside 1 basic block
- with branch prediction, only fetches & decode from the predicted branch

Speculation

- allows execution of instructions before their control dependencies are resolved
- increases ILP (over basic block boundaries)
- needs an mechanism to undo the effects

Speculation

Skip

need to separate forwarding from instruction completion

- values might be forwarded from speculated instructions to continue OoO execution
- but no speculated instruction should change the state of the program (in a way it can't be undone) or generate exceptions

commit instructions that are no longer speculative, in order

- commit = make the WB definitive (change program state)
- raise exception, if the case
- implemented with Reorder Buffer (ROB)
- ROB provides extra registers, in the same way as the reservation stations did

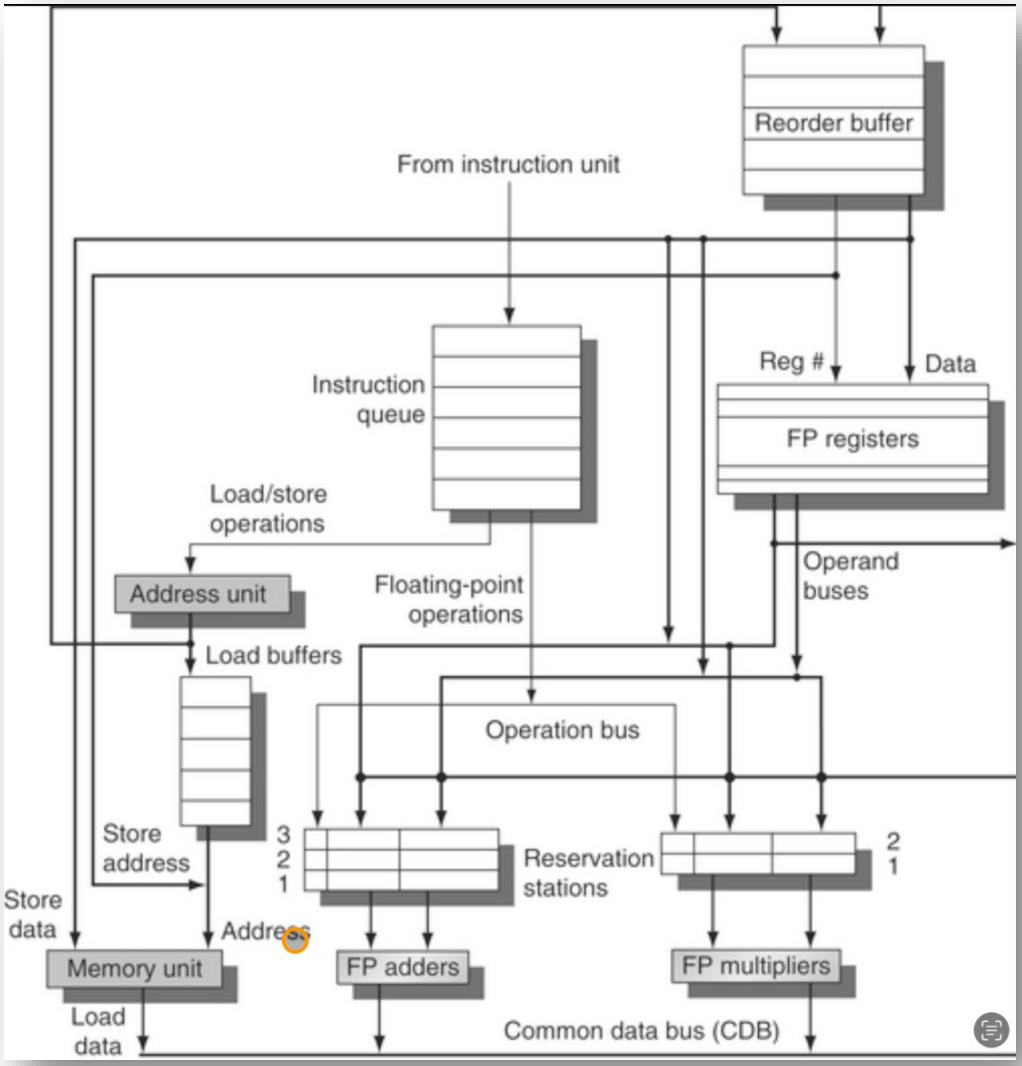
The Reorder Buffer (ROB)

Skip

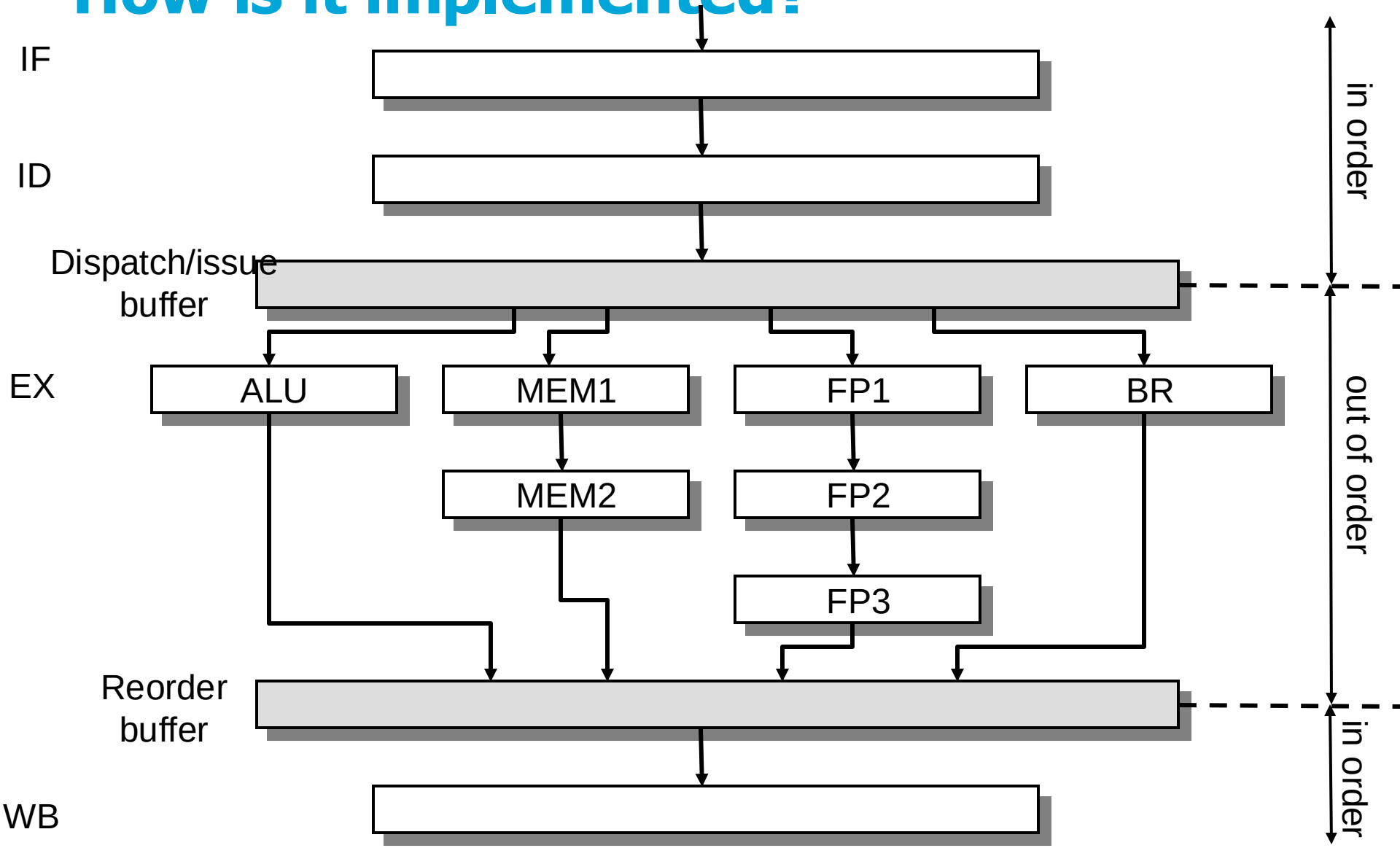
Writeback stage of Tomasulo writes to ROB instead of directly into register file (RF)

ROB data either commit to RF or used in reservation stations (RS)

Instruction issue stops when ROB is full



How is it implemented?



Summary

Skip

Modern processors exploit ILP to improve performance

Dynamically scheduled processors allow an instruction after a stalled instruction to proceed

- but instructions often commit in order to support precise exceptions
- decrease #stalls due to data hazards

Dynamic branch prediction

- decrease #stalls due to control hazards

Speculation

- increases ILP (over basic block boundaries)

Data (true) dependences are the only fundamental performance limitation

Register renaming removes false (name) dependences

Intermezzo – Side-story (END)

Skip

End – Branch Predictors

End Intermezzo - Branch Prediction

Returning to the ILP slides (6th Ed.)

Compiler Techniques for Exposing ILP

Pipeline scheduling

Separate dependent instruction from the source instruction by the pipeline latency of the source instruction

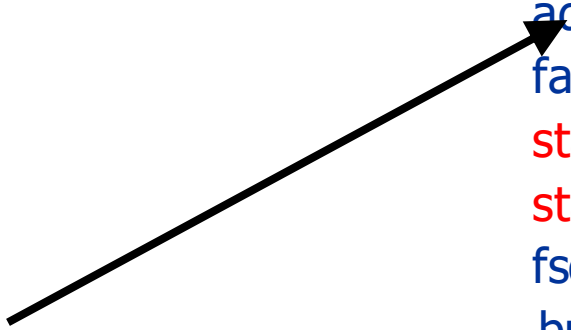
Example:

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Pipeline Stalls


```
Loop:   fld   f0,0(x1)
        stall
        fadd.d f4,f0,f2
        stall
        stall
        fsd f4,0(x1)
        addi x1,x1,-8
        bne x1,x2,Loop
```



```
Loop:   fld   f0,0(x1)
        addi x1,x1,-8
        fadd.d f4,f0,f2
        stall
        stall
        fsd f4,0(x1)
        bne x1,x2,Loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Loop Unrolling

Loop unrolling

- Unroll by a factor of 4 (assume # elements is divisible by 4)
- Eliminate unnecessary instructions

```
Loop:  fld f0,0(x1)
       fadd.d f4,f0,f2
       fsd f4,0(x1) //drop addi & bne
       fld f6,-8(x1)
       fadd.d f8,f6,f2
       fsd f8,-8(x1) //drop addi & bne
       fld f0,-16(x1)
       fadd.d f12,f0,f2
       fsd f12,-16(x1) //drop addi & bne
       fld f14,-24(x1)
       fadd.d f16,f14,f2
       fsd f16,-24(x1)
       addi x1,x1,-32
       bne x1,x2,Loop
```

- note: number of live registers vs. original loop

Loop Unrolling/Pipeline Scheduling

Pipeline schedule the unrolled loop:

```
Loop:   fld f0,0(x1)
        fld f6,-8(x1)
        fld f8,-16(x1)
        fld f14,-24(x1)
        fadd.d f4,f0,f2
        fadd.d f8,f6,f2
        fadd.d f12,f0,f2
        fadd.d f16,f14,f2
        fsd f4,0(x1)
        fsd f8,-8(x1)
        fsd f12,-16(x1)
        fsd f16,-24(x1)
        addi x1,x1,-32
        bne x1,x2,Loop
```

- 14 cycles
- 3.5 cycles per element

Strip Mining

Unknown number of loop iterations?

- Number of iterations = *n*
- Goal: make *k* copies of the loop body
- Generate pair of loops:
  - First executes *n* mod *k* times
  - Second executes *n* / *k* times
  - "Strip mining"

Dynamic Scheduling

Rearrange order of instructions to reduce stalls while maintaining data flow

Advantages:

- Compiler doesn't need to have knowledge of microarchitecture
- Handles cases where dependencies are unknown at compile time

Disadvantage:

- Substantial increase in hardware complexity
- Complicates exceptions (not discussed in slides)

Dynamic Scheduling

Dynamic scheduling implies:

- Out-of-order execution
- Out-of-order completion

Example 1:

```
fdiv.d f0,f2,f4
fadd.d f10,f0,f8
fsub.d f12,f8,f14
```

fsub.d is not dependent, issue before fadd.d

Dynamic Scheduling

Example 2:

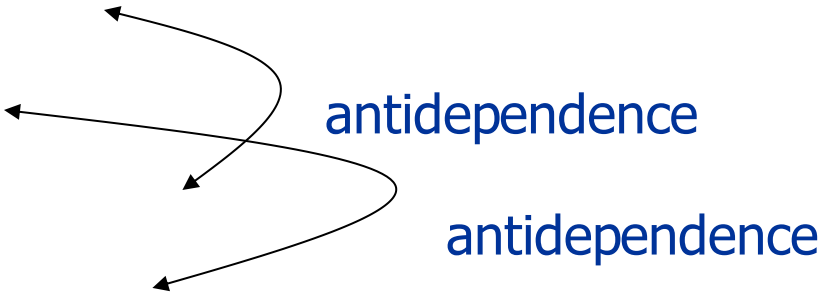
```
fdiv.d f0,f2,f4
fmul.d f6,f0,f8
fadd.d f0,f10,f14
```

fadd.d is not dependent, but the antidependence makes it impossible to issue earlier without register renaming

Register Renaming

Example 3:

fdiv.d f0,f2,f4
fadd.d f6,f0,f8
fsd f6,0(x1)
fsub.d f8,f10,f14
fmul.d f6,f10,f8



name dependence with f6

Register Renaming

Example 3:

fdiv.d f0,f2,f4

fadd.d S,f0,f8

fsd S,0(x1)

fsub.d T,f10,f14

fmul.d f6,f10,T

Now only RAW hazards remain, which can be strictly ordered

Register Renaming

Tomasulo's Approach

- Tracks when operands are available
- Introduces register renaming in hardware
  - Minimizes WAW and WAR hazards

Register renaming is provided by reservation stations (RS)

- Contains:
  - The instruction
  - Buffered operand values (when available)
  - Reservation station number of instruction providing the operand values

Register Renaming

RS fetches and buffers an operand as soon as it becomes available (not necessarily involving register file)

Pending instructions designate the RS to which they will send their output

- Result values broadcast on a result bus, called the common data bus (CDB)

Only the last output updates the register file

As instructions are issued, the register specifiers are renamed with the reservation station

May be more reservation stations than registers

Load and store buffers

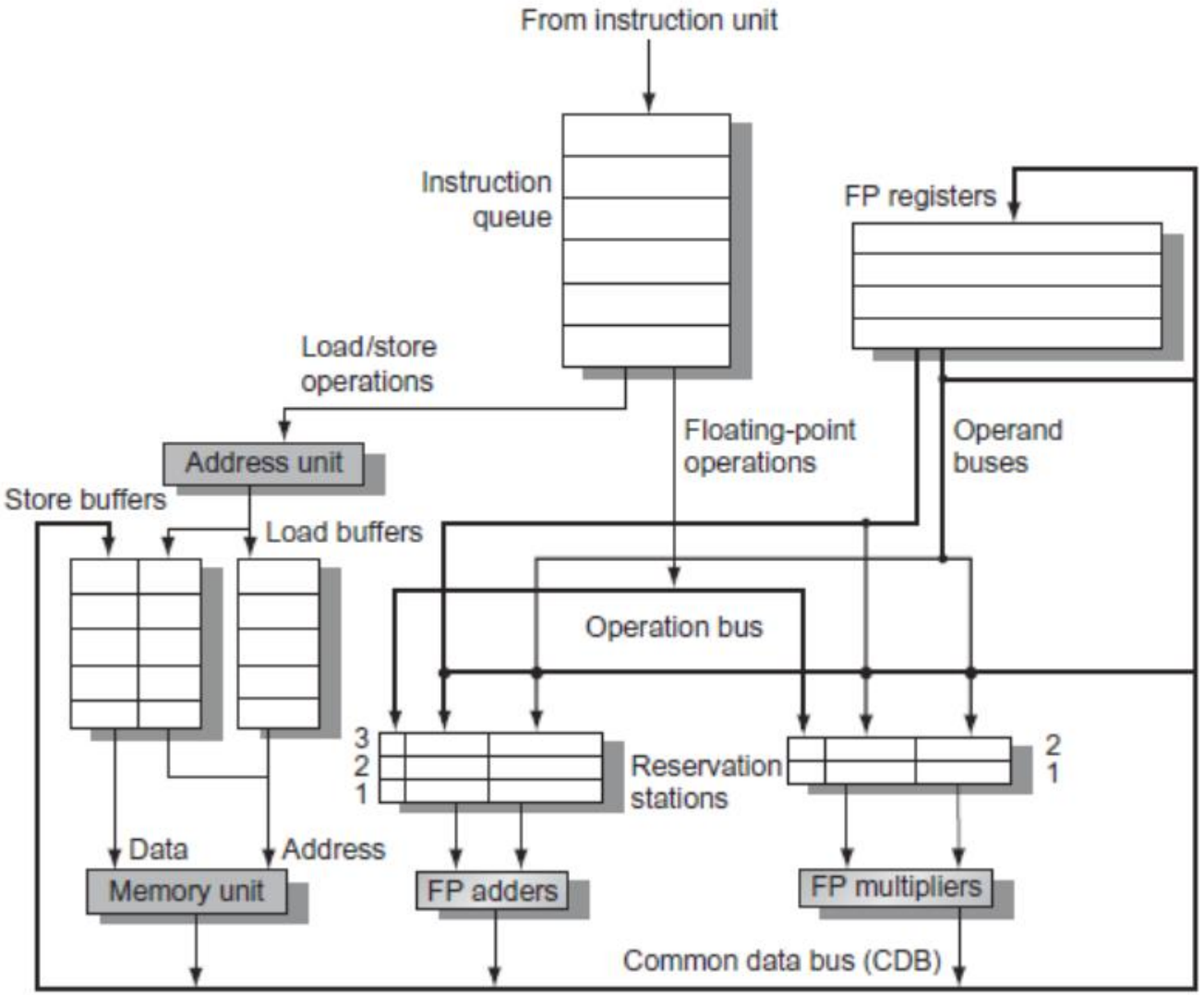
- Contain data and addresses, act like reservation stations

Tomasulo's Algorithm

Issue

Execute

Write result



Tomasulo's Algorithm

Three Steps:

- Issue
  - Get next instruction from FIFO queue
  - If available RS, issue the instruction to the RS with operand values if available
  - If operand values not available, stall the instruction
- Execute
  - When operand becomes available, store it in any reservation stations waiting for it
  - When all operands are ready, issue the instruction
  - Loads and store maintained in program order through effective address
  - No instruction allowed to initiate execution until all branches that proceed it in program order have completed
- Write result
  - Write result on CDB into reservation stations and store buffers
    - (Stores must wait until address and value are received)

Tomasulo alg: issue arithmetic ops

issue instruction:

if free reservation station (RS)

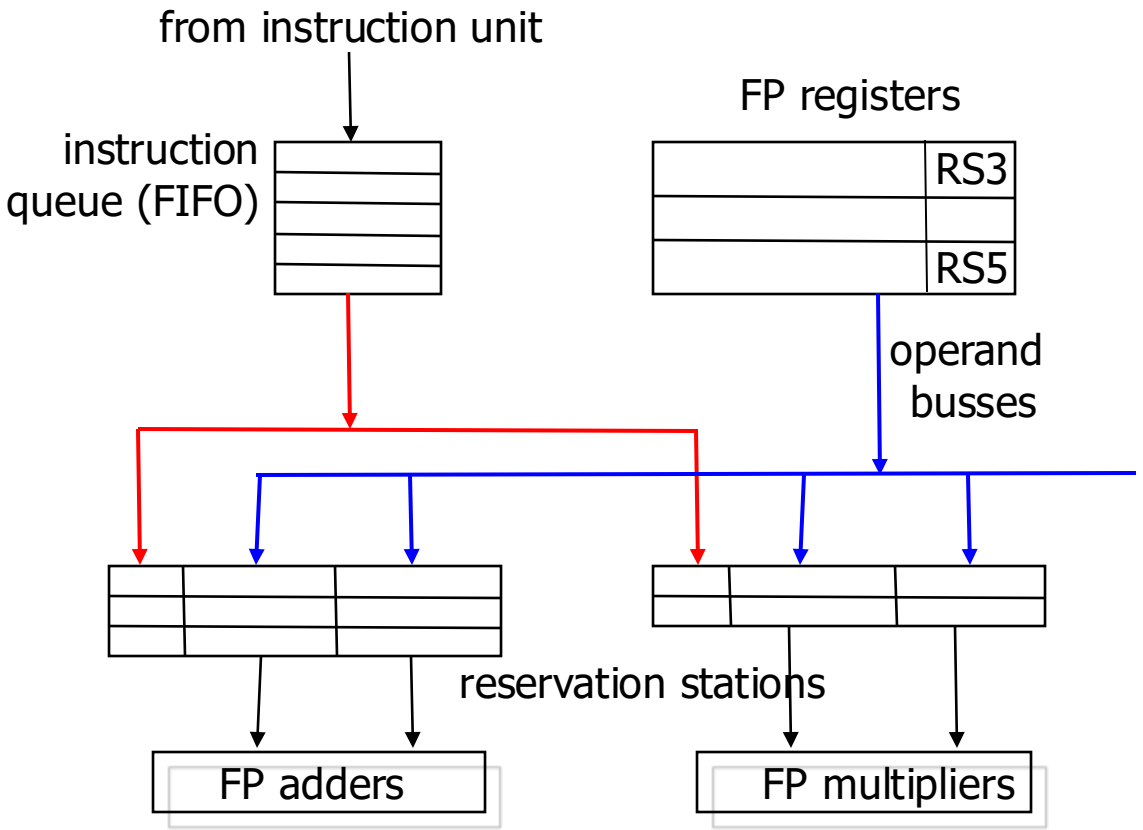
source operands

- in regs.: issue with values
- to-be-forwarded: link with the producer RS

destination operand

- link to a register
- each register F_i: RS# that has F_i as destination

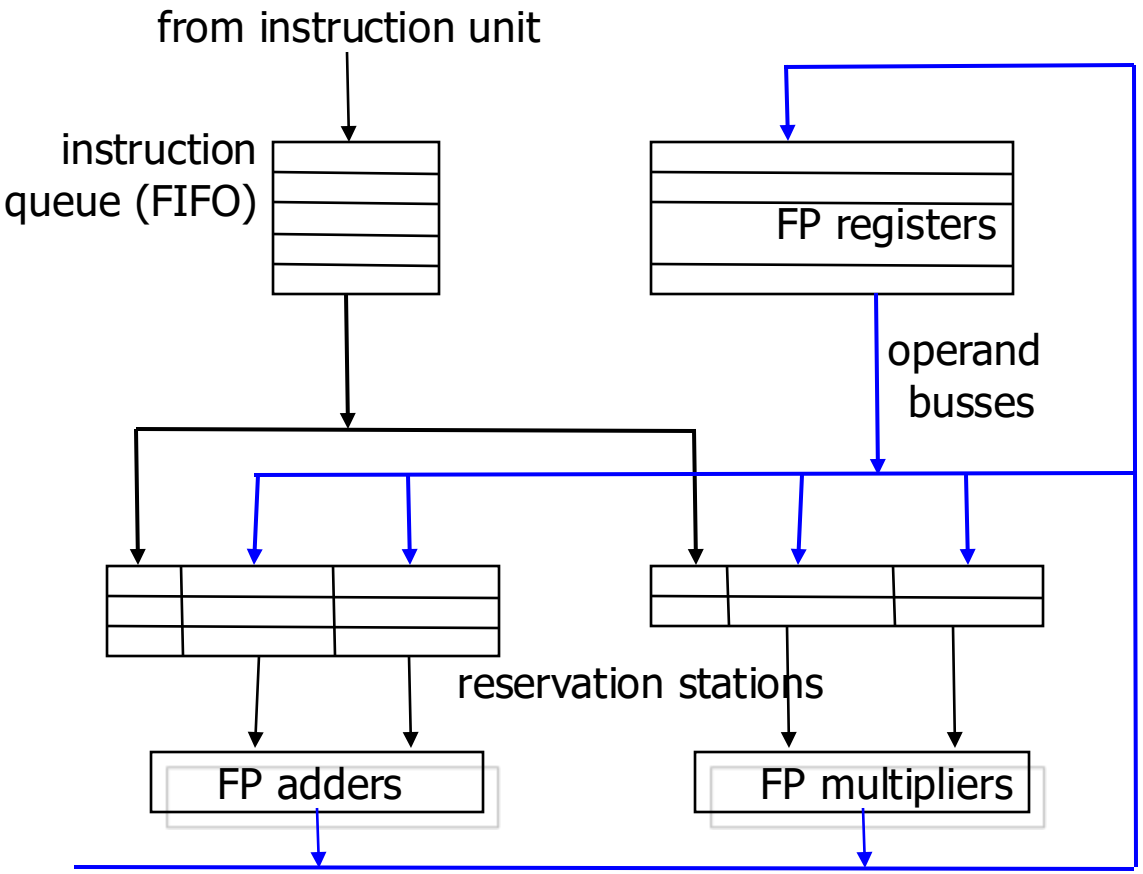
reservation station:



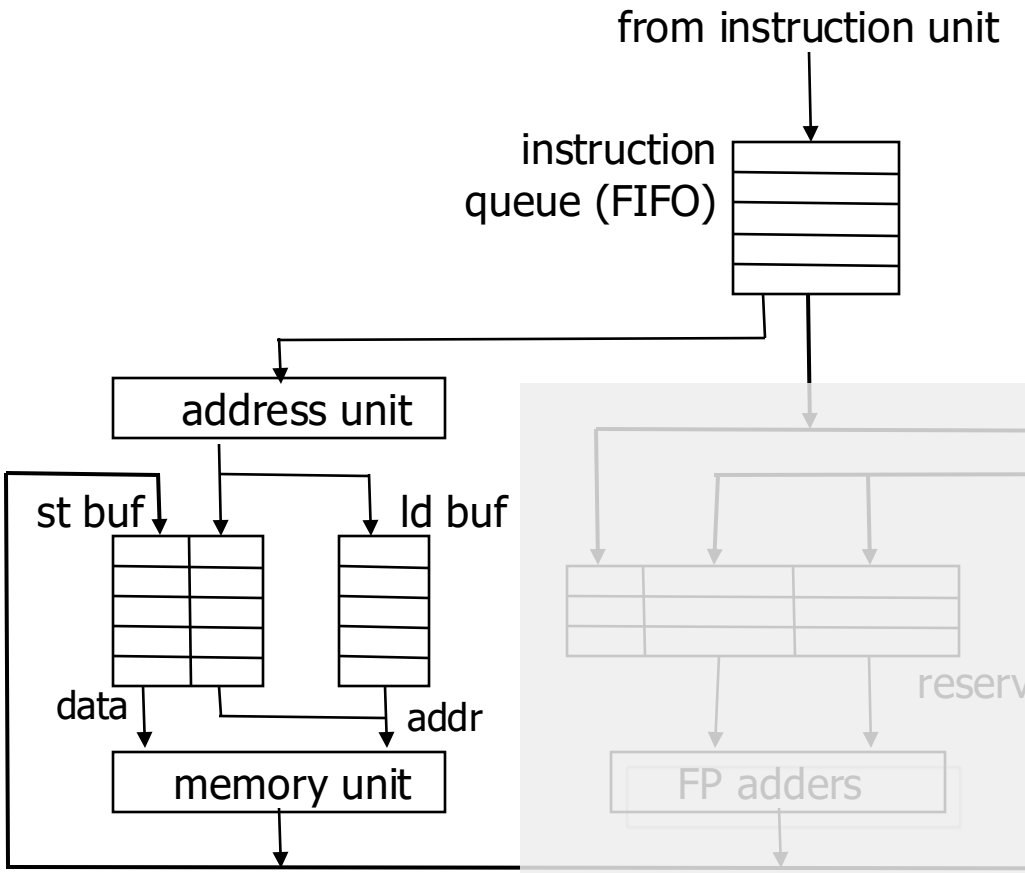
Opcode	RS producing op1	RS producing op2	value op1	value op2

Tomasulo alg: execute arithmetic ops

- execute when operands available (no RAW)
  - broadcasted on the data bus
  - from the data bus each RS (or register) that needs them, uses them



Tomasulo alg: issue&execute ld/st



issue load/store:

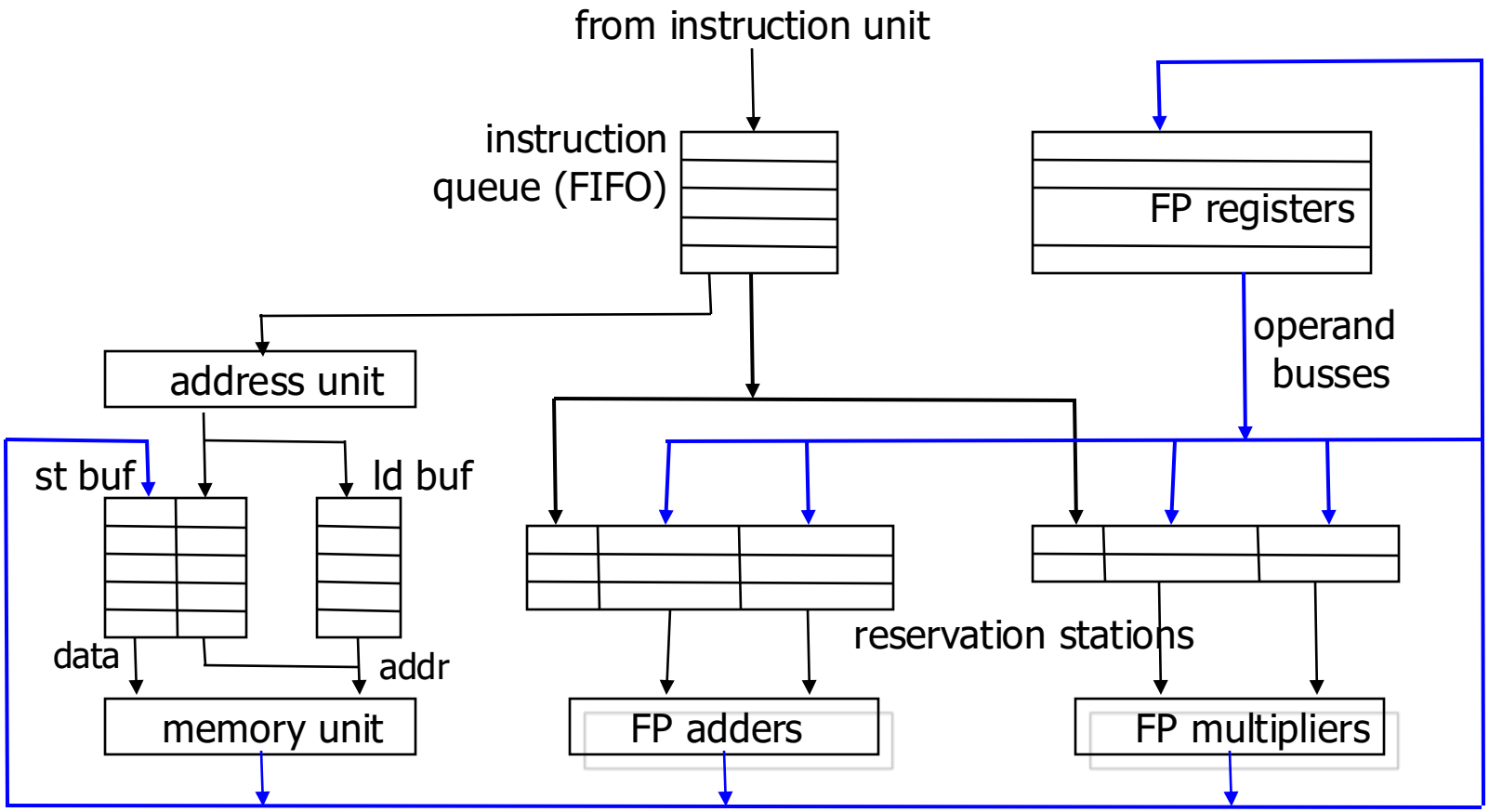
compute memory address (when base register available)

loads proceed as soon as the memory unit is free

stores wait for the value to be stored (from RS), if not immediate

loads and stores executed in order, to preserve correctness

Tomasulo alg: write back



when the result is available broadcasted on the data bus (to RS and regs)

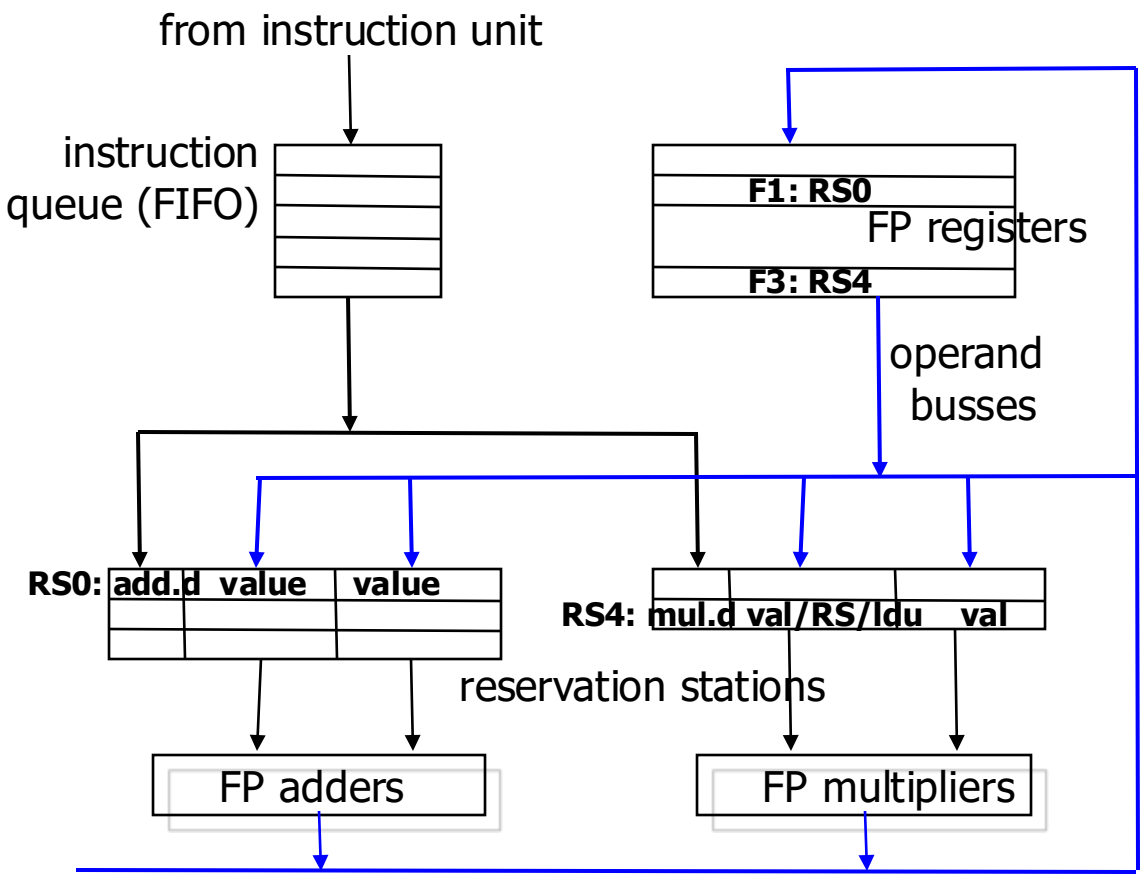
Tomasulo alg: WAR elimination

```
l.d F1, 0(R3)
…
mul.d F3, F1, F2
add.d F1, F4, F5
```

l.d completed: value copied from F1 to RS4 => mul.d & add.d independent

l.d not completed: mul.d waits the result of load unit ldu, (not register), add can execute (and write register) => mul.d & add.d independent

WAW elimination, similar



Tomasulo's out-of-order execution

register renaming – via reservation stations (RS)

- distributed RS: an operand can be forwarded to multiple RS in the same cycle (vs. a centralized register file: sequential access if only 1 port).
- elimination of WAR & WAW (RS+RF=virtual registers set)

forwarding/bypassing

- via common data bus, directly between the reservation stations

in-order issue/dispatch

out-of-order execution

out-of-order instruction retiring! (no precise exceptions)

Hardware-Based Speculation

Execute instructions along predicted execution paths but only commit the results if prediction was correct

Instruction commit: allowing an instruction to update the register file when instruction is no longer speculative

Need an additional piece of hardware to prevent any irrevocable action until an instruction commits

I.e. updating state or taking an execution

Reorder Buffer

Reorder buffer – holds the result of instruction between completion and commit

Four fields:

- Instruction type: branch/store/register
- Destination field: register number
- Value field: output value
- Ready field: completed execution?

Modify reservation stations:

- Operand source is now reorder buffer instead of functional unit

Reorder Buffer

Issue:

Allocate RS and ROB, read available operands

Execute:

Begin execution when operand values are available

Write result:

Write result and ROB tag on CDB

Commit:

When ROB reaches head of ROB, update register

When a mispredicted branch reaches head of ROB, discard all entries

Reorder Buffer

Register values and memory values are not written until an instruction commits

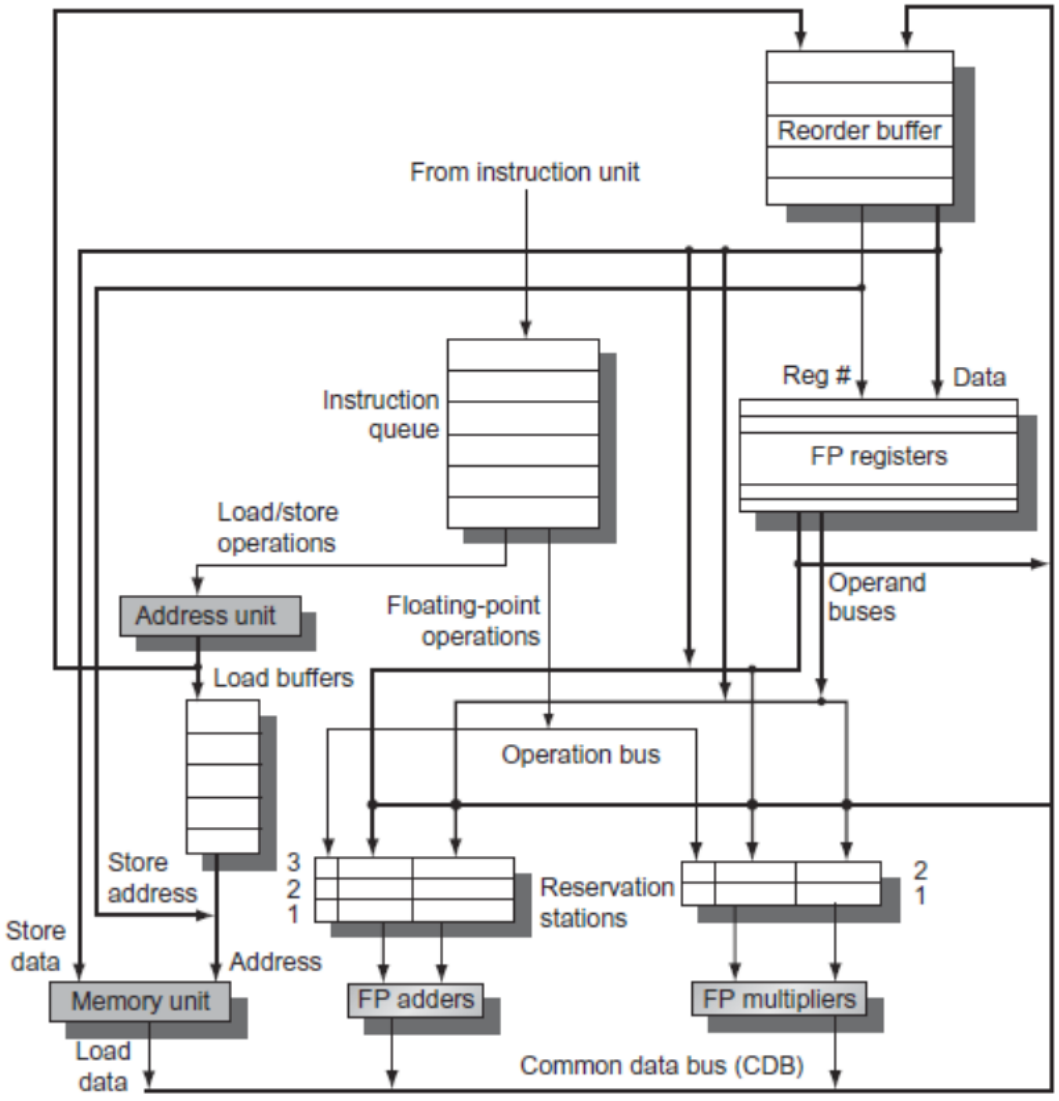
On misprediction:

- Speculated entries in ROB are cleared

Exceptions:

- Not recognized until it is ready to commit

Reorder Buffer



Reorder Buffer

	Reorder buffer					
Entry	Busy	Instruction		State	Destination	Value
1	No	fld	f6,32(x2)	Commit	f6	Mem[32 + Regs[x2]]
2	No	fld	f2,44(x3)	Commit	f2	Mem[44 + Regs[x3]]
3	Yes	fmul.d	f0,f2,f4	Write result	f0	#2 × Regs[f4]
4	Yes	fsub.d	f8,f2,f6	Write result	f8	#2 – #1
5	Yes	fdiv.d	f0,f0,f6	Execute	f0	
6	Yes	fadd.d	f6,f8,f2	Write result	f6	#4 + #2

	Reservation stations							
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	fmul.d	Mem[44 + Regs[x3]]	Regs[f4]			#3	
Mult2	Yes	fdiv.d		Mem[32 + Regs[x2]]	#3		#5	

	FP register status									
Field	f0	f1	f2	f3	f4	f5	f6	f7	f8	f10
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	…	Yes	Yes

End – Multiple Issue

Multiple Issue and Static Scheduling

To achieve CPI < 1, need to complete multiple instructions per clock

Solutions:

- Statically scheduled superscalar processors
- VLIW (very long instruction word) processors
- Dynamically scheduled superscalar processors

Multiple Issue

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

VLIW Processors

Some considerations

These are additional slides (not from the book)!

Some Background (from Digital Logic Design course)

Think back to "digital logic design"

Data vs. Control Subsystem

Data Subsystem → Control "points" to direct the data flow and operations

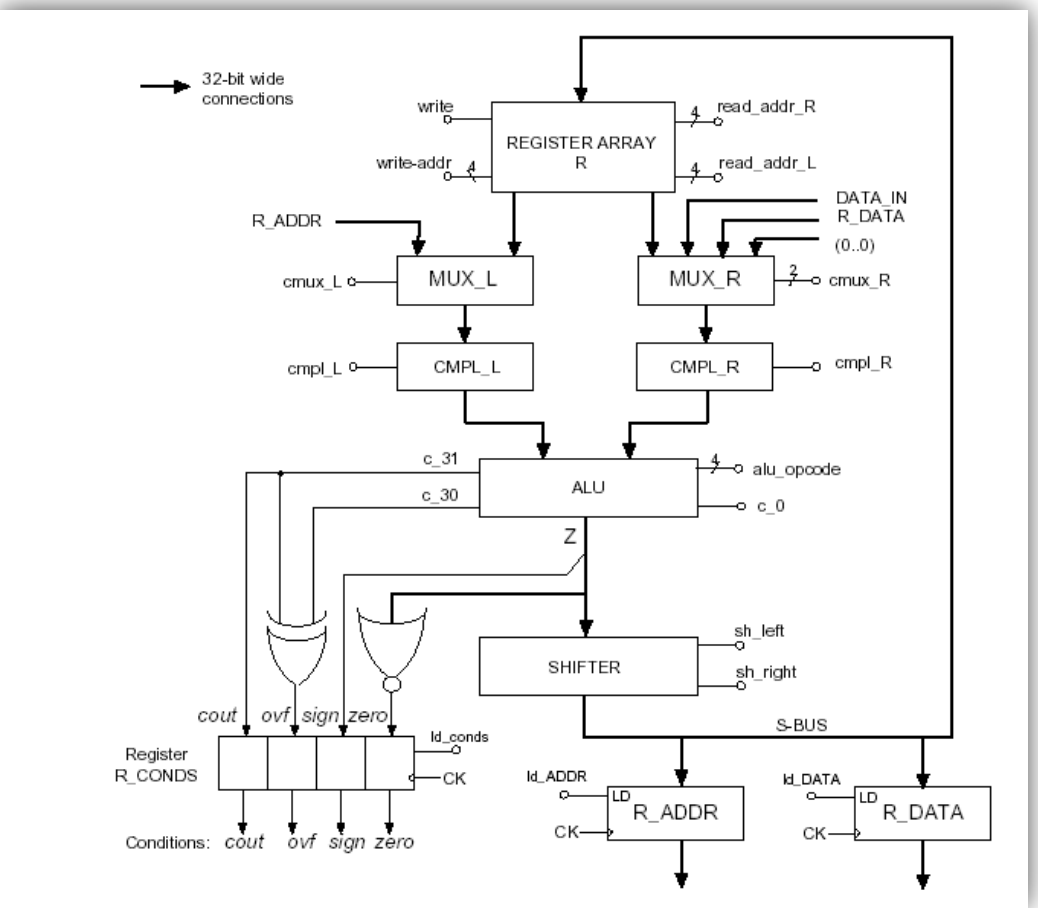
Control Subsystem → Microprogrammed Controller

Microinstructions – Vertical vs. Horizontal

Data subsystem

Data subsystem performs all data operations **controlled by** *control signals*

Where are the control signals coming from?

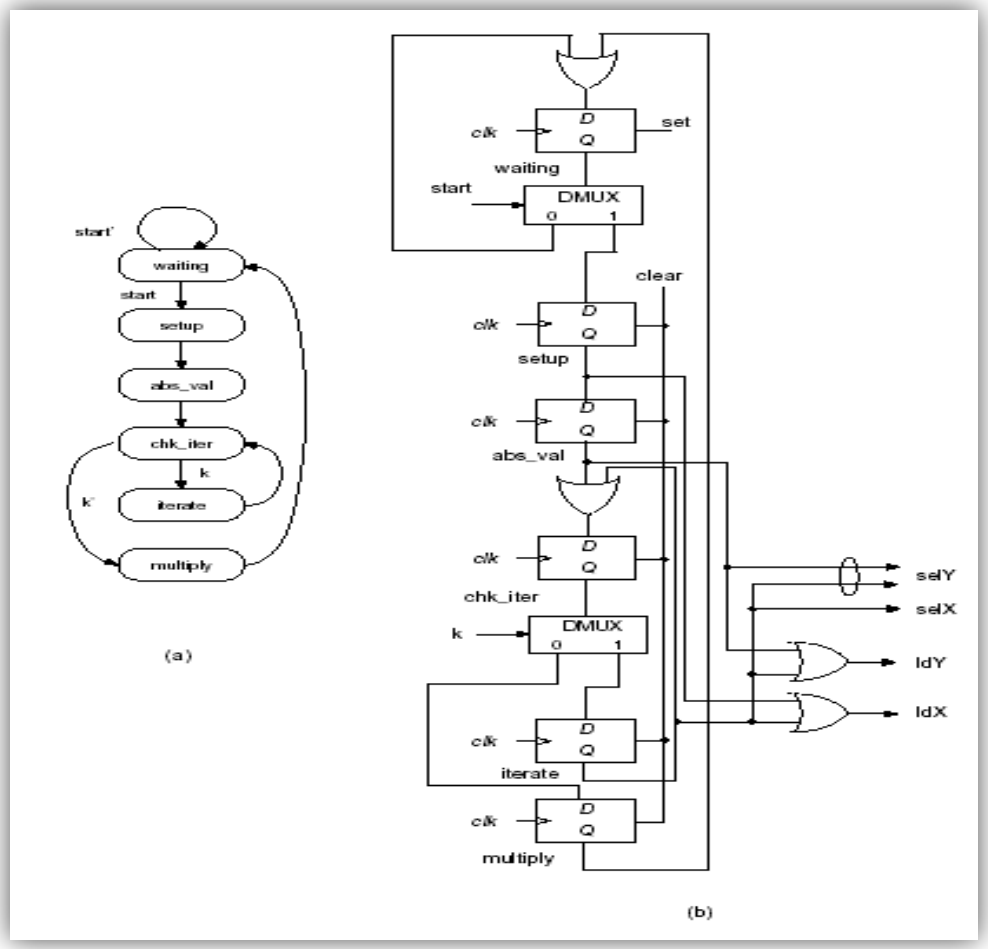


Control subsystem

Control subsystem generates all necessary control signals for the data subsystem to function correctly

Usually, it is an FSM with each state generating control signals

How can we abstract from this view?

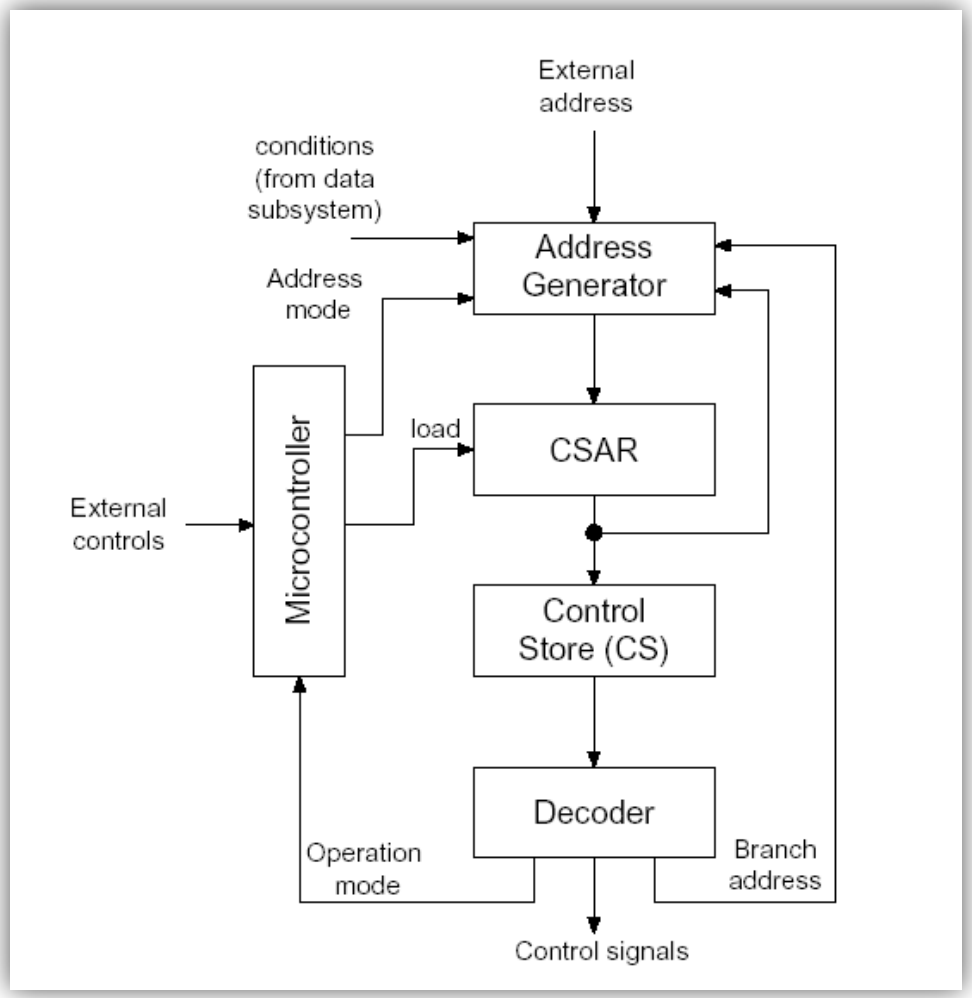


Microprogrammed Controller

Use microprograms to implement the generation of control signals

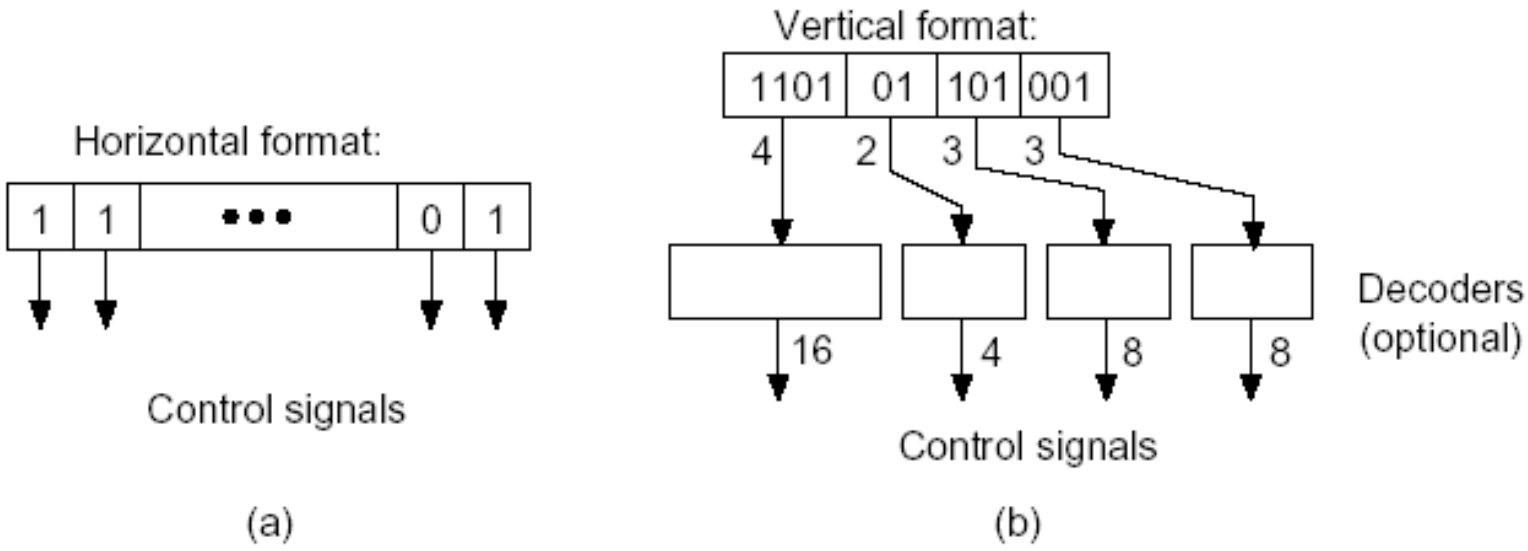
How do we "pack" the control signals into a microinstruction?

(concept actually from a 1951 paper by Wilkes)



Control field

- HORIZONTAL (unpacked, decoded)
- VERTICAL (packed, encoded)



Why Microprogramming?

Hardware was not yet "sophisticated" (in the 70s)

- ~2300 transistors (Intel 4004) to ~29000 transistors (Intel 8088)
- modern-day processor: several billions of transistors

Wish to separate ISA design from technology (constraints)!

- ISA determines functionality of processor from user/compiler point-of-view
- Technology determines basic operations → controlled via microcode

Clear mismatch – what is the solution?

- Break down complex (ISA) instructions into simpler micro-operations → microcode
- (microcode still exists – compatibility and chip I/O)

Consequence – (new) microcode needed for each processor generation

Why not make them visible to the programmer/compiler?

From Microcode to RISC and VLIW

Vertical microcode:

- Sequential simple hardware operations
- Decoding needed in order to save control store space

Making vertical microcode visible to the programmer → RISC processors

Horizontal microcode:

- Parallel control of operations, i.e., parallel operations
- No decoding needed

Making horizontal microcode visible to the programmer → VLIW processors

VLIW Processors (A Short History)

Roots: Alan Turing (1946) and Maurice Wilkes (1951, horizontal microcode)

Joseph Fisher (1979) working on translating sequential code to horizontal microcode

Multiflow (by Fisher, 1984) and Cydrome (by Bob Rau, 1984) started their business and produced actual products → expertise ended up in HP Labs

Media processing reintroduced VLIW, e.g., Chromatic MPact and Philips Trimedia, but more commercially successful was Texas Instruments C6X series

Transmeta founded 2000 – Crusoe VLIW processors to emulate x86 via a software layer and Code Morphing technology

Intel started their IA-64 and worked on the Itanium series

Nowadays, many DSP-related functionalities, e.g., in mobile phones, are performed by VLIW processors – this means one or more VLIW processors sold for each ARM processor sold

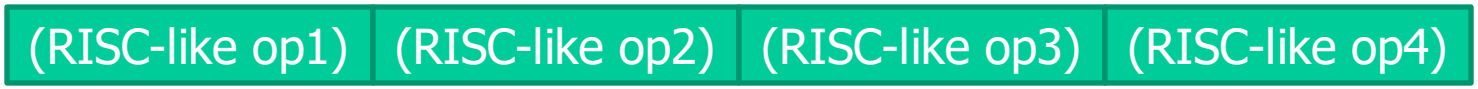
ILP – Issue more instructions per cycle

Statically scheduled superscalar processors – in-order

Dynamically scheduled superscalar processors – out-of-order

Very Large Instruction Word processors

Statically schedule multiple instructions per cycle, e.g.,



In contrast to superscalar:

No hardware needed to detect and issue multiple instructions

Simpler instruction fetch and decode hardware – more power efficient

Mainly found in DSP/embedded environments

Design considerations of VLIW processor

Determine the amount of parallelism – number of execution slots

Determine the execution units per slot

Schedule code – by hand of compiler?

Execute code – in real hardware or in simulator?

QUESTION: when is a VLIW most effective?

VLIW Processors (Characteristics)

Parallel execution units – instruction word specify execution in issue slots

Relies on compiler to schedule parallel instructions --> local (within single basic block) vs. global scheduling, loop unrolling, software pipelining, trace scheduling, predicated execution

Simple hardware means more power efficient → in the end: energy eff.

VLIW are excellent for applications with a lot of parallelism, e.g., highly parallel kernels that are executed for a large portion of time

VLIW Processors (Disadvantages)

High number of NOPs when issue slots are unmatched or executing inherently low ILP code

Code size increase (storage of NOPs) – considering that in certain scenarios 50% of power can lie in I-cache alone, this can be quite significant; code compression can overcome this, but also requires power for decode

Code compatibility limited between processor generations – superscalar processors can run legacy code without recompilation, but also needs recompilation to exploit processor enhancements

High pressure on register file for wide VLIW processors (# of ports)

Lockstep operation – e.g., halting when cache misses

Mechanisms to overcome these disadvantages can consume a lot of power

VLIW Processors (Advantages)

Use when application domain is known and has inherent parallelism

When done right:

- Power/Energy efficient
- High performance
- Area efficient

Slides from the 6th Ed.

Continuing the story from the book!

VLIW Processors

Package multiple operations into one instruction

Example VLIW processor:

- One integer instruction (or branch)
- Two independent floating-point operations
- Two independent memory references

Must be enough parallelism in code to fill the available slots

VLIW Processors

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
fld f0,0(x1)	fld f6,-8(x1)			
fld f10,-16(x1)	fld f14,-24(x1)			
fld f18,-32(x1)	fld f22,-40(x1)	fadd.d f4,f0,f2	fadd.d f8,f6,f2	
fld f26,-48(x1)		fadd.d f12,f0,f2	fadd.d f16,f14,f2	
		fadd.d f20,f18,f2	fadd.d f24,f22,f2	
fsd f4,0(x1)	fsd f8,-8(x1)	fadd.d f28,f26,f24		
fsd f12,-16(x1)	fsd f16,-24(x1)			addi x1,x1,-56
fsd f20,24(x1)	fsd f24,16(x1)			
fsd f28,8(x1)				bne x1,x2,Loop

Disadvantages:

- Statically finding parallelism
- Code size
- No hazard detection hardware
- Binary code compatibility

Dynamic Scheduling, Multiple Issue, and Speculation

Modern microarchitectures:

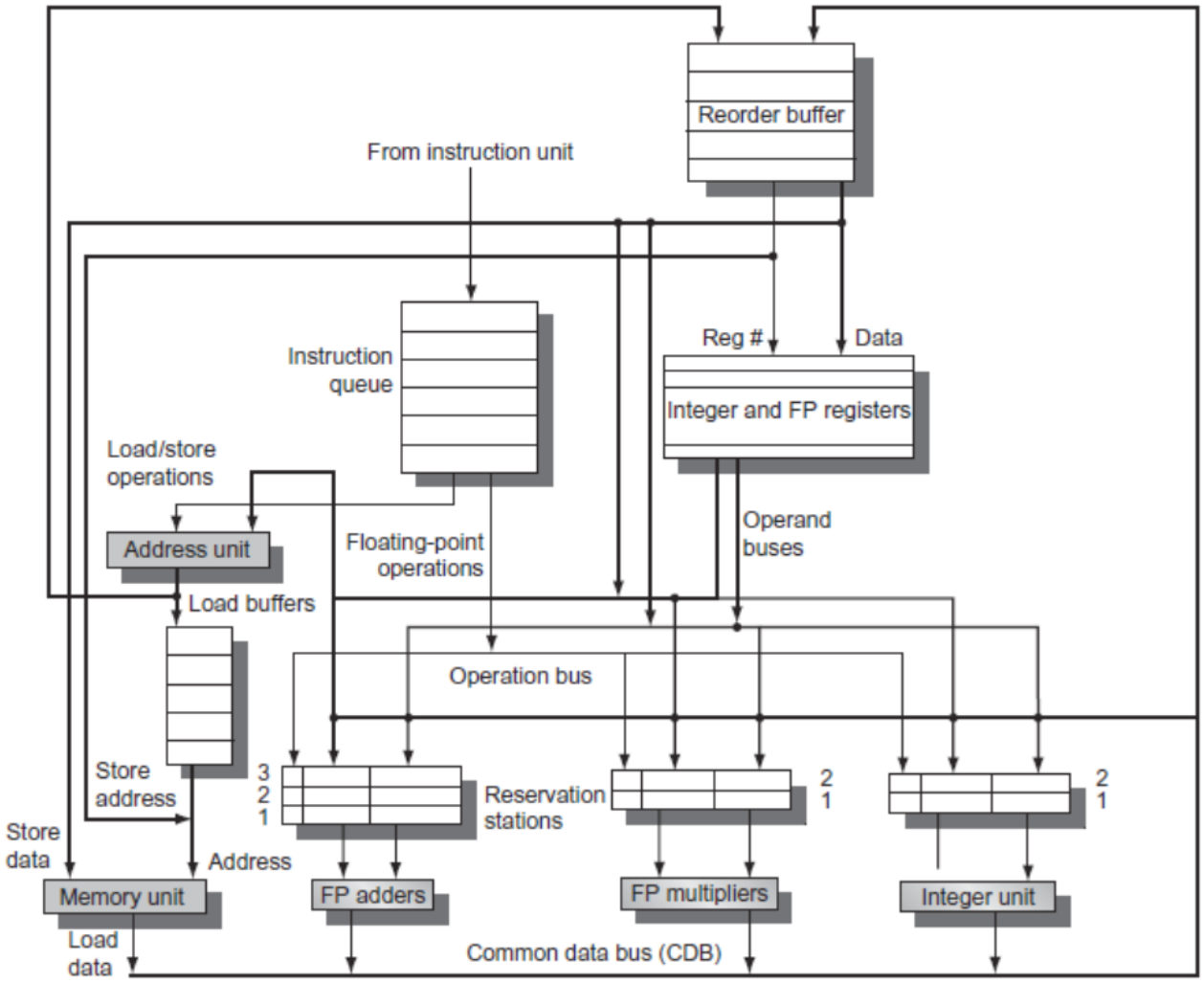
- Dynamic scheduling + multiple issue + speculation

Two approaches:

- Assign reservation stations and update pipeline control table in half clock cycles
  - Only supports 2 instructions/clock
- Design logic to handle any possible dependencies between the instructions

Issue logic is the bottleneck in dynamically scheduled superscalars

Overview of Design



Multiple Issue

Examine all the dependencies amoung the instructions in the bundle

If dependencies exist in bundle, encode them in reservation stations

Also need multiple completion/commit

To simplify RS allocation:

- Limit the number of instructions of a given class that can be issued in a "bundle", i.e. on FP, one integer, one load, one store

Branch Target Buffer

Is NOT branch prediction!

Try to predict the **target address** of taken branches, unconditional branches, jumps – put this in a **buffer**

Q: Why is this interesting? What problem is being "solved"?

Hint 1/Q: related to I-cache or D-cache?

Hint 2/Q: what can we do when we "know" or can predict the target address?

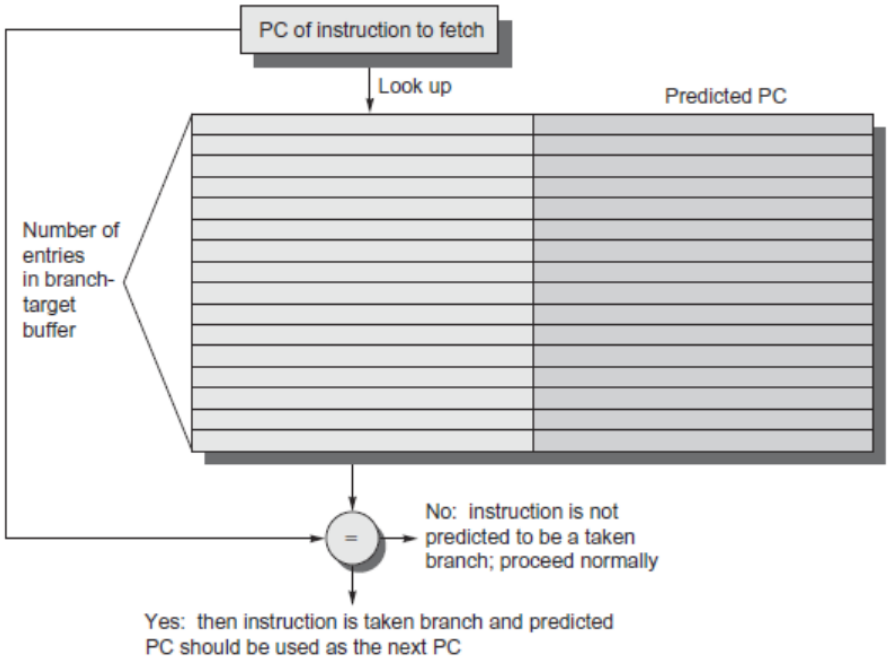
A: We can now prefetch instructions (in the I-cache)!!

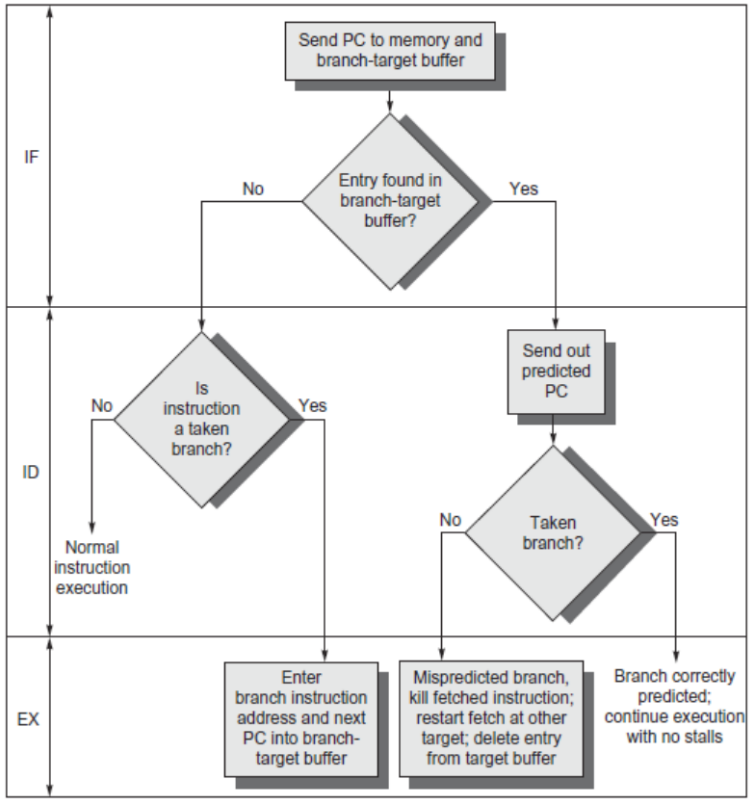
Branch-Target Buffer

Need high instruction bandwidth

Branch-Target buffers

Next PC prediction buffer, indexed by current PC





Branch Folding

Optimization:

- Larger branch-target buffer
- Add target instruction into buffer to deal with longer decoding time required by larger buffer
- "Branch folding"

Return Address Predictor

Q: What does "return" mean in this case?

A: Return from procedure call

Easy to predict if always the same location (in code) is calling procedures!

Q: What if many different locations are calling the same procedure? How to predict in this scenario?

A: Instead of prediction, use **return stack** !

Push (return address = PC+4) when calling procedure & Pop (return address) when returning

Note: also works well for nested procedure calls!

Return Address Predictor

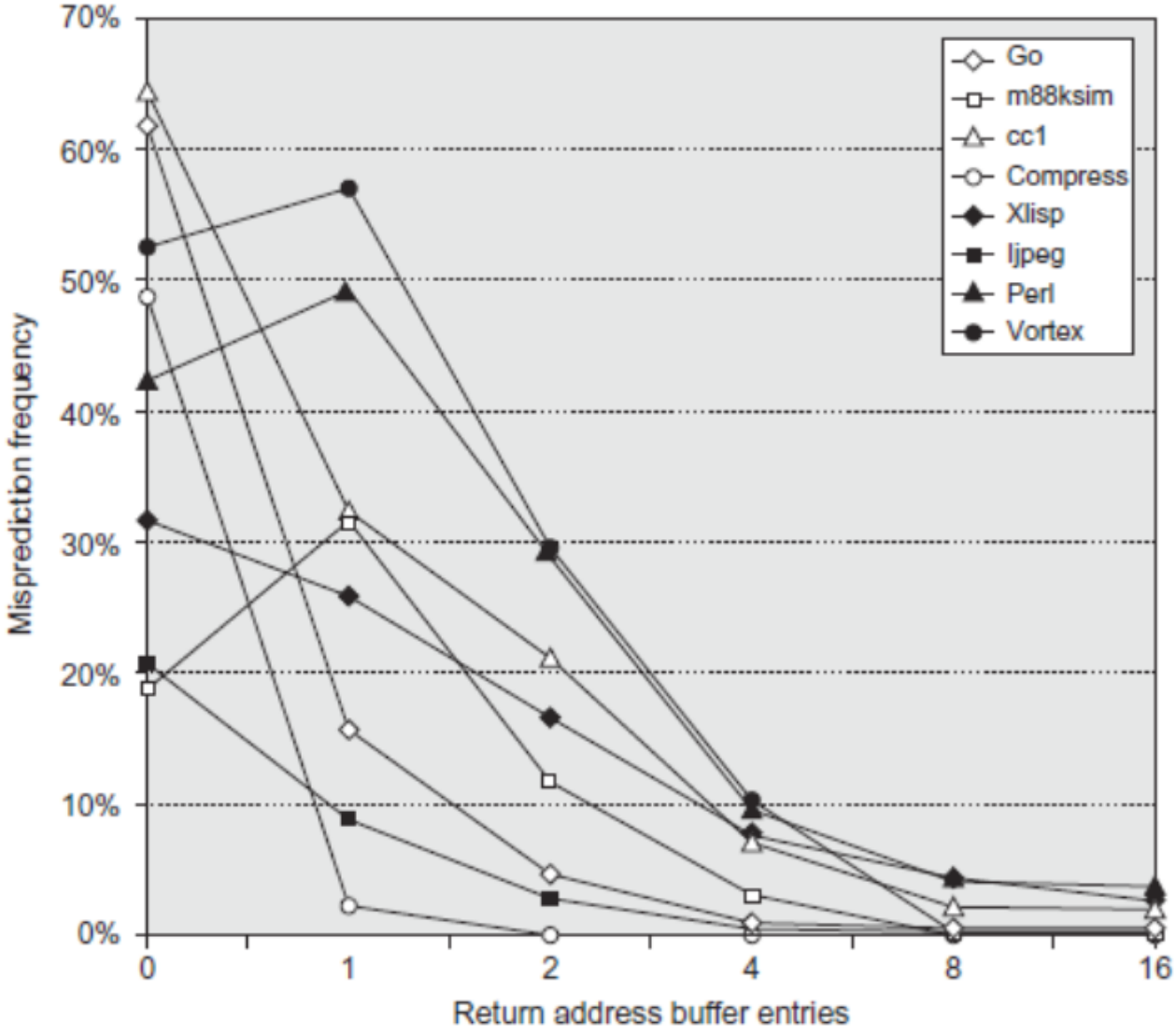
Most unconditional branches come from function returns

The same procedure can be called from multiple sites

- Causes the buffer to potentially forget about the return address from previous calls

Create return address buffer organized as a stack

Return Address Predictor



Integrated Instruction Fetch Unit

Design monolithic unit that performs:

- Branch prediction
- Instruction prefetch
  - Fetch ahead
- Instruction memory access and buffering
  - Deal with crossing cache lines

Register Renaming

- Register renaming vs. reorder buffers
  - Instead of virtual registers from reservation stations and reorder buffer, create a single register pool
    - Contains visible registers and virtual registers
  - Use hardware-based map to rename registers during issue
  - WAW and WAR hazards are avoided
  - Speculation recovery occurs by copying during commit
  - Still need a ROB-like queue to update table in order
  - Simplifies commit:
    - Record that mapping between architectural register and physical register is no longer speculative
    - Free up physical register used to hold older value
    - In other words: SWAP physical registers on commit
  - Physical register de-allocation is more difficult
    - Simple approach: deallocate virtual register when next instruction writes to its mapped architecturally-visibly register

Integrated Issue and Renaming

Combining instruction issue with register renaming:

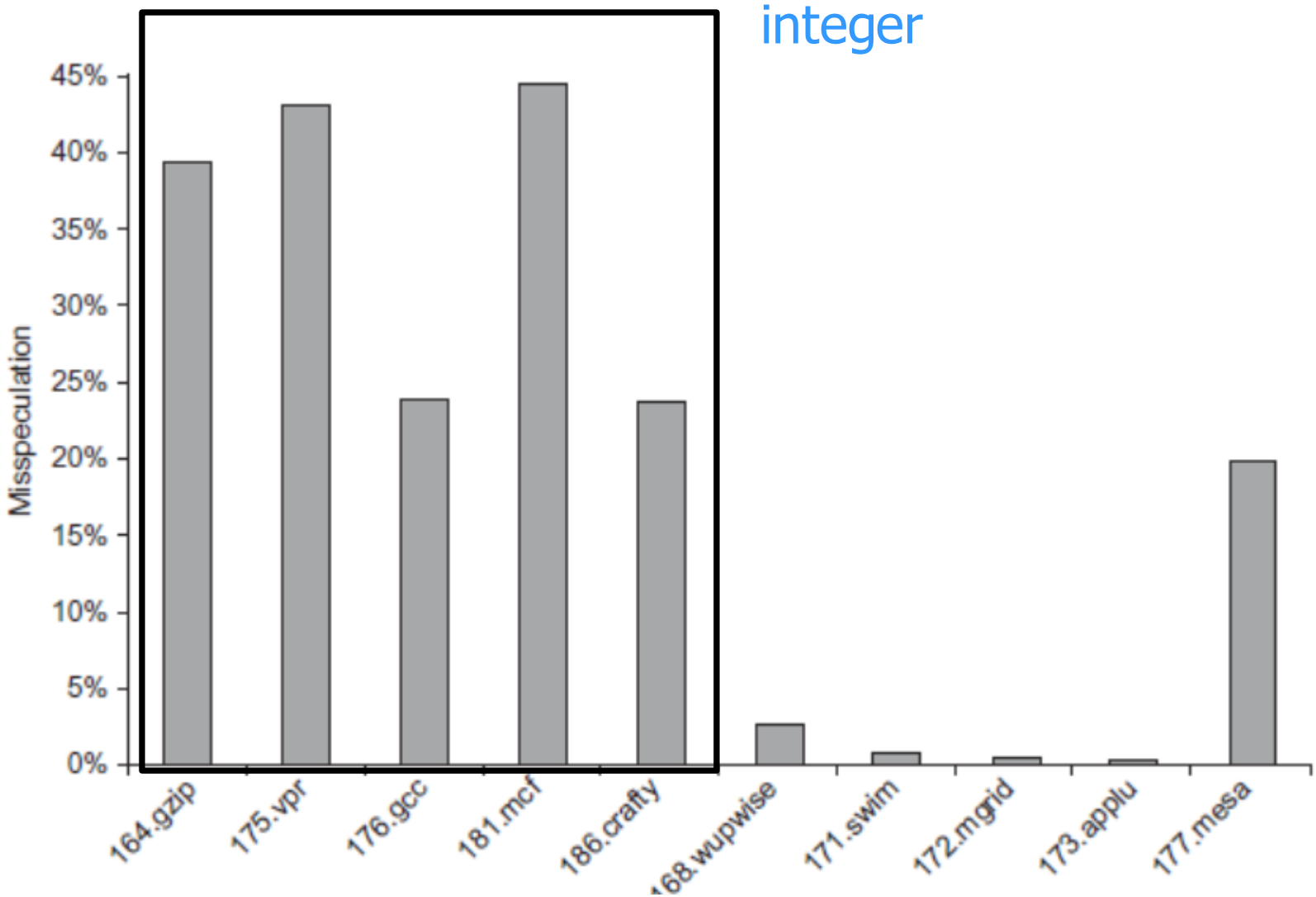
- Issue logic pre-reserves enough physical registers for the bundle
- Issue logic finds dependencies within bundle, maps registers as necessary
- Issue logic finds dependencies between current bundle and already in-flight bundles, maps registers as necessary

Instr. #	Instruction	Physical register assigned or destination	Instruction with physical register numbers	Rename map changes
1	add x1,x2,x3	p32	add p32,p2,p3	x1-> p32
2	sub x1,x1,x2	p33	sub p33,p32,p2	x1->p33
3	add x2,x1,x2	p34	add p34,p33,x2	x2->p34
4	sub x1,x3,x2	p35	sub p35,p3,p34	x1->p35
5	add x1,x1,x2	p36	add p36,p35,p34	x1->p36
6	sub x1,x3,x1	p37	sub p37,p3,p36	x1->p37

How Much?

- How much to speculate
  - Mis-speculation degrades performance and power relative to no speculation
    - May cause additional misses (cache, TLB)
  - Prevent speculative code from causing higher costing misses (e.g. L2)
- Speculating through multiple branches
  - Complicates speculation recovery
- Speculation and energy efficiency
  - Note: speculation is only energy efficient when it significantly improves performance

How Much?


integer
Misspeculation
45%
40%
35%
30%
25%
20%
15%
10%
5%
0%
164.gzip
175.vpr
176.gcc
181.mcf
186.crafty
168.wupwise
171.swim
172.mgrid
173.applu
177.mesa

Energy Efficiency

Value prediction

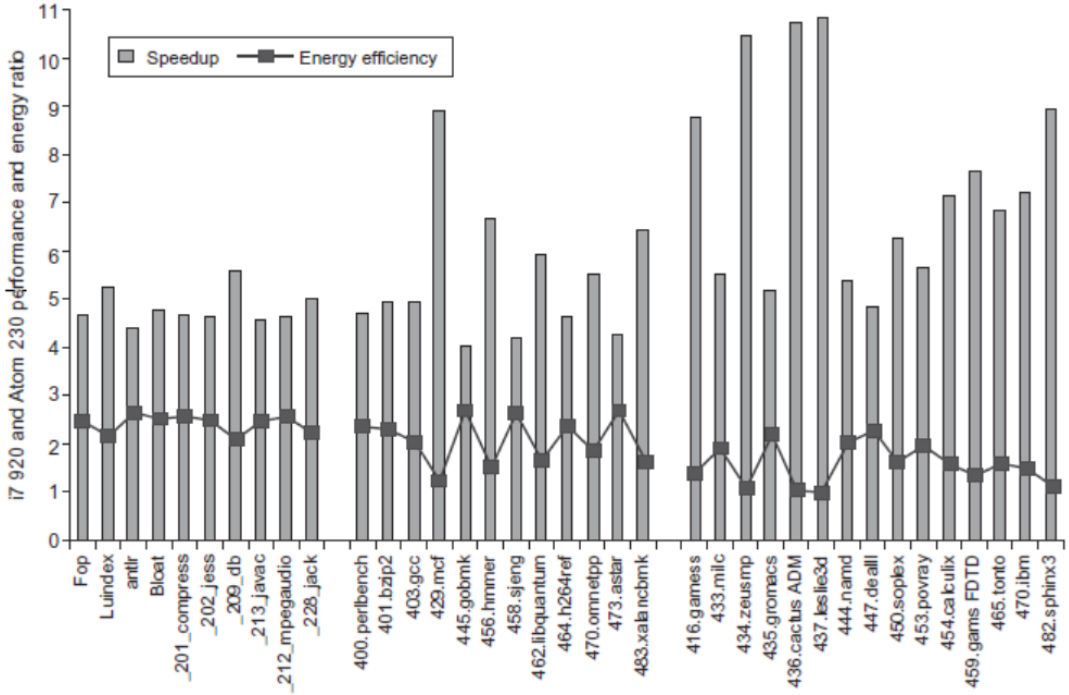
- Uses:
  - Loads that load from a constant pool
  - Instruction that produces a value from a small set of values
- Not incorporated into modern processors
- Similar idea--*address aliasing prediction*--is used on some processors to determine if two stores or a load and a store reference the same address to allow for reordering

Fallacies and Pitfalls

Fallacy = common misbelief

Pitfall = easily made mistake

F: It is easy to predict the performance/energy efficiency of two different versions of the same ISA if we hold the technology constant



Fallacies and Pitfalls

F: Processors with lower CPIs / faster clock rates will also be faster

Processor	Implementation technology	Clock rate	Power	SPECCInt2006 base	SPECCFP2006 baseline
Intel Pentium 4 670	90 nm	3.8 GHz	115 W	11.5	12.2
Intel Itanium 2	90 nm	1.66 GHz	104 W approx. 70 W one core	14.5	17.3
Intel i7 920	45 nm	3.3 GHz	130 W total approx. 80 W one core	35.5	38.4

Itanium had same CPI, lower clock

Fallacies and Pitfalls

P: Sometimes bigger and dumber is better

Pentium 4 and Itanium were advanced designs, but could not achieve their peak instruction throughput because of relatively small caches as compared to i7

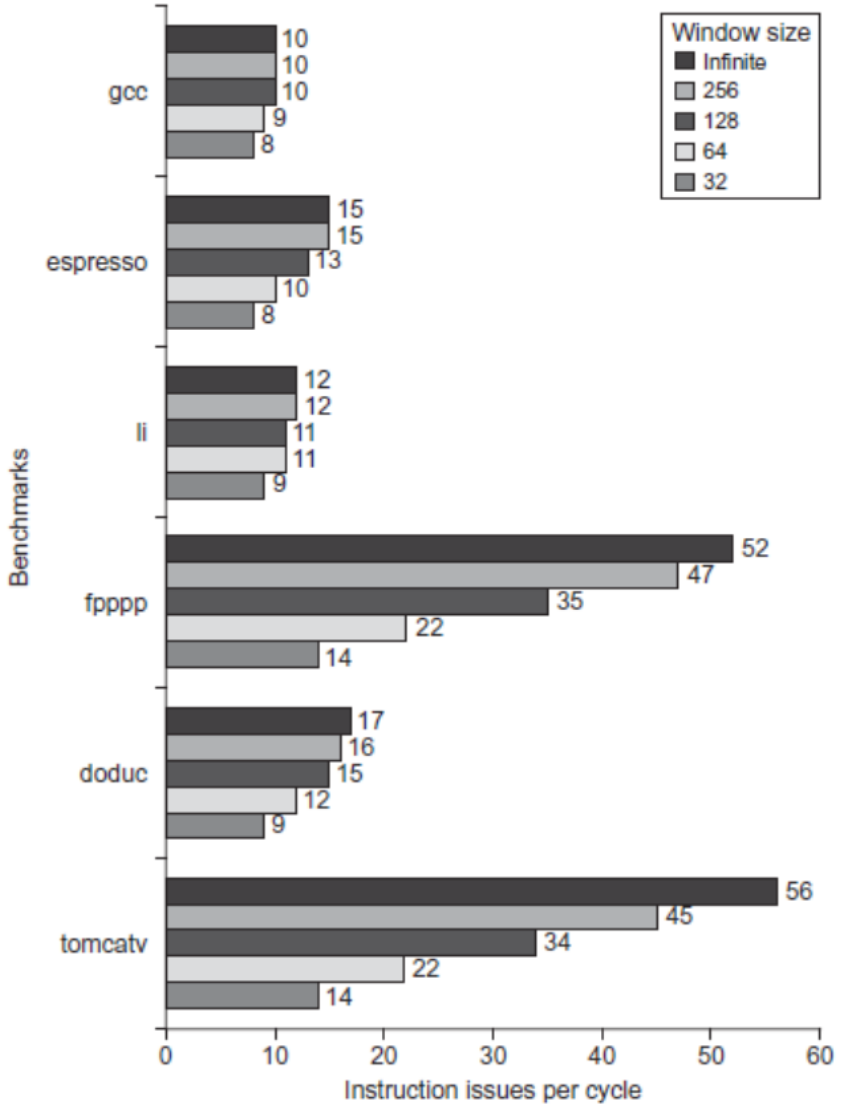
P: And sometimes smarter is better than bigger and dumber

TAGE branch predictor outperforms gshare with less stored predictions

Fallacies and Pitfalls

P: Believing that there are large amounts of ILP available, if only we had the right techniques

Read the book: there are many "ideal" or complex hardware assumptions – thus, almost unrealistic – generating these results.



END of NEW CH3