Lecture 5: EITF20 Computer Architecture

Anders Ardö

EIT – Electrical and Information Technology, Lund University

November 19, 2014

Outline

Instruction Level Parallelism - ILP

ILP: Overlap execution of unrelated instructions: **Pipelining**
Two main approaches:

- DYNAMIC $\Longrightarrow$ hardware detects parallelism
- STATIC $\Longrightarrow$ software detects parallelism

Often a mix between both.

Pipeline CPI = Ideal CPI + Structural stalls
+ Data hazard stalls + Control stalls

Why loop unrolling works

- Longer sequences of straight code without branches (longer basic blocks) allows for ***easier compiler static rescheduling***
- Longer basic blocks also facilitates dynamic rescheduling such as Scoreboard and Tomasulo's algorithm

Dynamic Branch Prediction

- Branches limit performance because:
  - Branch penalties
  - Limit to available Instruction Level Parallelism
- Solution: ***Dynamic branch prediction*** to predict the outcome of conditional branches.
  Benefits:
  - Reduce the time to when the branch condition is known
  - Reduce the time to calculate the branch target address

Dependencies

- Two instructions must be ***independent*** in order to execute in parallel
- There are three general types of dependencies that limit parallelism:
  - Data dependencies
  - Name dependencies
  - Control dependencies
- Dependencies are properties of the ***program***
- Whether a dependency leads to a hazard or not is a property of the ***pipeline implementation***

Scoreboard pipeline

- Goal of scoreboarding is to maintain an execution rate of one instruction per clock cycle by executing an instruction as early as possible.
- Instructions execute out-of-order when there are sufficient resources and no data dependencies.
- A scoreboard is a hardware unit that keeps track of
  - the instructions that are in the process of being executed,
  - the functional units that are doing the executing,
  - and the registers that will hold the results of those units.
- A scoreboard centrally performs all hazard detection and resolution and thus controls the instruction progression from one step to the next.

Summary

- ILP:
  - Rescheduling and loop unrolling are important to take advantage of potential Instruction Level Parallelism
- Dynamic instruction scheduling
  - An alternative to compile-time scheduling
  - Does not need recompilation to increase performance
  - Used in most new processor implementations
- Dynamic Branch Prediction
  - reduce branch penalties by early prediction of conditional branch outcomes

Questions!

QUESTIONS?

COMMENTS?

Lecture 5 agenda

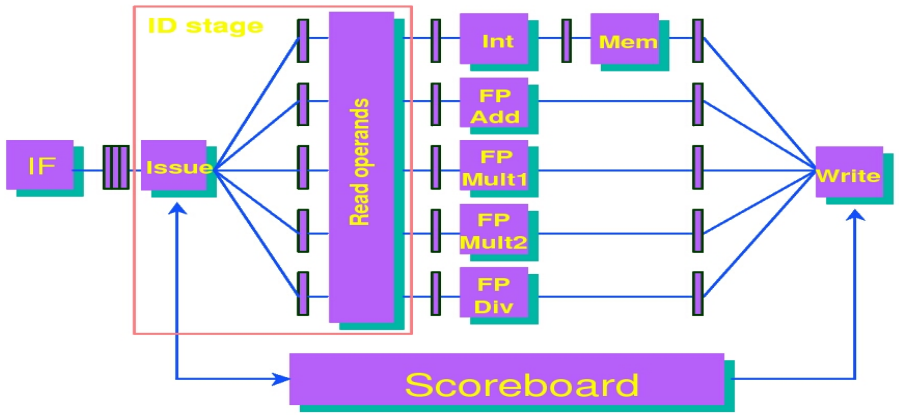
Chapters 2.4-2.8, 3.1-3.4 in "Computer Architecture"

1. Reiteration
2. Dynamic scheduling - Tomasulo
3. Superscalar, VLIW
4. Speculation
5. ILP limitations
6. What we have done so far

Outline

Scoreboard pipeline

- **Issue**: Decode and check for structural hazards
- **Read operands**: wait until no data hazards, then read operands
- All data hazards are handled by the scoreboard



Limitations with Scoreboard

- The number of scoreboard entries (*window size*)
- The number and types of functional units
- **Number of datapaths to registers**
- **The presence of name dependencies**

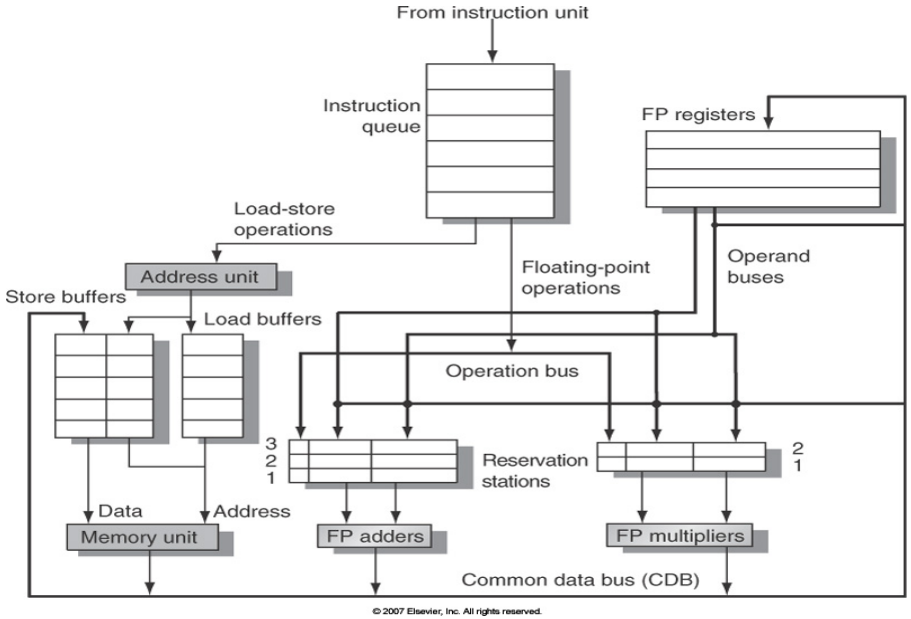
Tomasulo's algorithm addresses the last two limitations.

Tomasulo's Algorithm

Another dynamic instruction scheduling algorithm

- For IBM 360/91, a few years after the CDC 6600 (Scoreboard)
- Goal: High performance without compiler support
- Differences between Tomasulo & Scoreboard:
  - Control & Buffers *distributed* with FUs (called ***reservation stations***) vs. centralized in Scoreboard
  - Register names in instructions replaced by pointers to reservation station buffer (***HW register renaming***)
  - Common Data Bus broadcasts results to all FUs
  - Loads and Stores treated as FUs as well

Tomasulo Organization



Three Stages of Tomasulo Alg.

- 1. ***Issue*** – get instruction from FP Op Queue
  - If reservation station free (no structural hazard), the instruction is issued together with its operands (renames registers)
- 2. ***Execution*** – operate on operands (EX)
  - When both operands are ready, then execute; if not ready, watch Common Data Bus (CDB) for operands (snooping)
- 3. ***Write result*** – finish execution (WB)
  - Write on CDB to all awaiting functional units; mark reservation station available
- Normal bus: data + destination
- Common Data Bus: data + ***source*** (snooping)

Tomasulo example, cycle 0

Instruction status

Instruction	j	k	Issue	Exec. compl.	Write result
LD F6	34+	R2			
LD F2	45+	R3			
MULTD F0	F2	F4			
SUBD F8	F6	F2			
DIVD F10	F0	F6			
ADDD F6	F8	F2			

Load buffers

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Functional unit status

Time	Name	Busy	Op	src 1 Vj	src 2 Vk	RS for j Qj	RS for k Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU								

Clock: 0

Tomasulo example, cycle 1

Instruction status

Instruction	j	k	Issue	Exec. compl.	Write result
LD F6	34+	R2	1		
LD F2	45+	R3			
MULTD F0	F2	F4			
SUBD F8	F6	F2			
DIVD F10	F0	F6			
ADDD F6	F8	F2			

Time

Load buffers

	Busy	Address
Load1	Yes	R2+34
Load2	No	
Load3	No	

Functional unit status

Time	Name	Busy	Op	src 1 Vj	src 2 Vk	RS for j Qj	RS for k Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU				Load1				

Clock: 1

Tomasulo example, cycle 3

Instruction status

Instruction	j	k	Issue	Exec. compl.	Write result
LD F6	34+	R2	1	3	
LD F2	45+	R3	2		
MULTD F0	F2	F4	3		
SUBD F8	F6	F2			
DIVD F10	F0	F6			
ADDD F6	F8	F2			

Time

	Busy	Address
Load1	Yes	R2+32
Load2	Yes	R3+45
Load3	No	

Functional unit status

Time	Name	Busy	Op	src 1 Vj	src 2 Vk	RS for j Qj	RS for k Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Mult		F4	Load2	
	Mult2	No					

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1	Load2		Load1				

Clock: 3

Tomasulo example, cycle 4

Instruction status

Instruction	j	k	Issue	Exec. compl.	Write result
LD F6	34+	R2	1	3	4
LD F2	45+	R3	2	4	
MULTD F0	F2	F4	3		
SUBD F8	F6	F2	4		
DIVD F10	F0	F6			
ADDD F6	F8	F2			

Time

	Busy	Address
Load1	No	
Load2	Yes	R3+45
Load3	No	

Functional unit status

Time	Name	Busy	Op	src 1 Vj	src 2 Vk	RS for j Qj	RS for k Qk
	Add1	Yes	Sub	M(R2+34)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	Mult		F4	Load2	
	Mult2	No					

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1	Load2		-	Add1			

Clock: 4

Tomasulo example, cycle 5

Instruction status

Instruction		j	k	Issue	Exec. compl.	Write result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

Time

Load buffers

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Functional unit status

Time	Name	Busy	Op	src 1 Vj	src 2 Vk	RS for j Qj	RS for k Qk
2	Add1	Yes	Sub	M(R2+34)	M(R3+45)		-
	Add2	No					
	Add3	No					
10	Mult1	Yes	Mult	M(R3+45)	F4	-	
	Mult2	Yes	Div		F6	Mult1	

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1	-			Add1	Mult2		

Clock: 5

Tomasulo example, cycle 7

Instruction status

Instruction		j	k	Issue	Exec. compl.	Write result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

Time

Load buffers

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Functional unit status

Time	Name	Busy	Op	src 1 Vj	src 2 Vk	RS for j Qj	RS for k Qk
0	Add1	Yes	Sub	M(R2+34)	M(R3+45)		
	Add2	Yes	Add		F2	Add1	
	Add3	No					
8	Mult1	Yes	Mult	M(R3+45)	F4		
	Mult2	Yes	Div		F6	Mult1	

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1			Add2	Add1	Mult2		

Clock: 7

Tomasulo example, cycle 10

Instruction status

Instruction		j	k	Issue	Exec. compl.	Write result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	

Time

Load buffers

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Functional unit status

Time	Name	Busy	Op	src 1 Vj	src 2 Vk	RS for j Qj	RS for k Qk
	Add1	No					
0	Add2	Yes	Add	F6-F2	F2		
	Add3	No					
5	Mult1	Yes	Mult	M(R3+45)	F4		
	Mult2	Yes	Div		F6	Mult1	

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1			Add2		Mult2		

Clock: 10

Elimination of WAR hazards

Example:

```
LD      F6, 34(R2)
...     ...
DIVD    F10,F0,F6
ADDD    F6,F8,F2
```

- ADDD can safely finish before **DIVD** has read register F6 because:
  - DIVD has ***renamed*** register F6 to point at the reservation station
  - LD ***broadcasts*** its result on the Common Data Bus
- ***Register renaming*** can thus be done:
  - statically by the compiler
  - dynamically by the hardware

Tomasulo example, cycle 11

Instruction status

Instruction		j	k	Issue	Exec. compl.	Write result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

Time

Load buffers

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Functional unit status

Time	Name	Busy	Op	src 1 Vj	src 2 Vk	RS for j Qj	RS for k Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	Mult	M(R3+45)	F4		
	Mult2	Yes	Div		F6	Mult1	

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	Mult1			-		Mult2		

Clock: 11

Tomasulo example, cycle 16

Instruction status

Instruction		j	k	Issue	Exec. compl.	Write result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

Time

Load buffers

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Functional unit status

Time	Name	Busy	Op	src 1 Vj	src 2 Vk	RS for j Qj	RS for k Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	Div	F0	F6	-	

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU	-					Mult2		

Clock: 16

Tomasulo example, cycle 57

Instruction status

Instruction		j	k	Issue	Exec. compl.	Write result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	57
ADDD	F6	F8	F2	6	10	11

Time

Load buffers

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Functional unit status

Time	Name	Busy	Op	src 1 Vj	src 2 Vk	RS for j Qj	RS for k Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status

	F0	F2	F4	F6	F8	F10	...	F30
FU						-		

Clock: 57

Benefits Tomasulo

- distributed hazard detection logic
  - distributed reservation stations
  - Common Data Bus (CDB) with snooping
- elimination WAR,WAW hazards (renaming registers)

Dynamic scheduling Tomasulo - summary

- tolerates unpredictable delays
- compile for one pipeline - run effectively on another
- significant increase in HW complexity
- out-of-order execution, completion
- register renaming

Outline

Getting CPI < 1!

- Issuing multiple instructions per clock cycle
  - ***Superscalar***: varying number of instructions/cycle (1-8) scheduled by compiler or HW
    - IBM Power5, Pentium 4, Sun SuperSparc, DEC Alpha
    - Simple hardware, complicated compiler or...
    - Very complex hardware but simple for compiler
  - ***Very Long Instruction Word*** (VLIW): fixed number of instructions (3-5) scheduled by the compiler
    - HP/Intel IA-64, Itanium
    - Simple hardware, difficult for compiler
      high performance through extensive compiler optimization

Approaches for multiple issue

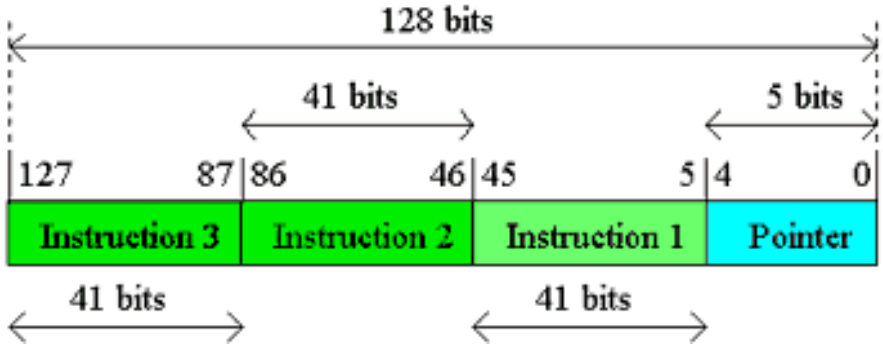
	Issue	**Hazard detection**	**Scheduling**	**Characteristics /examples**
Superscalar	dynamic	HW	static	in-order execution ARM
Superscalar	dynamic	HW	dynamic	out-of-order execution
Superscalar	dynamic	HW	dynamic	speculation Pentium 4 IBM power5
VLIW	static	compiler	static	TI C6x
EPIC	static	compiler	mostly static	Itanium

Very Long Instruction Word (VLIW)

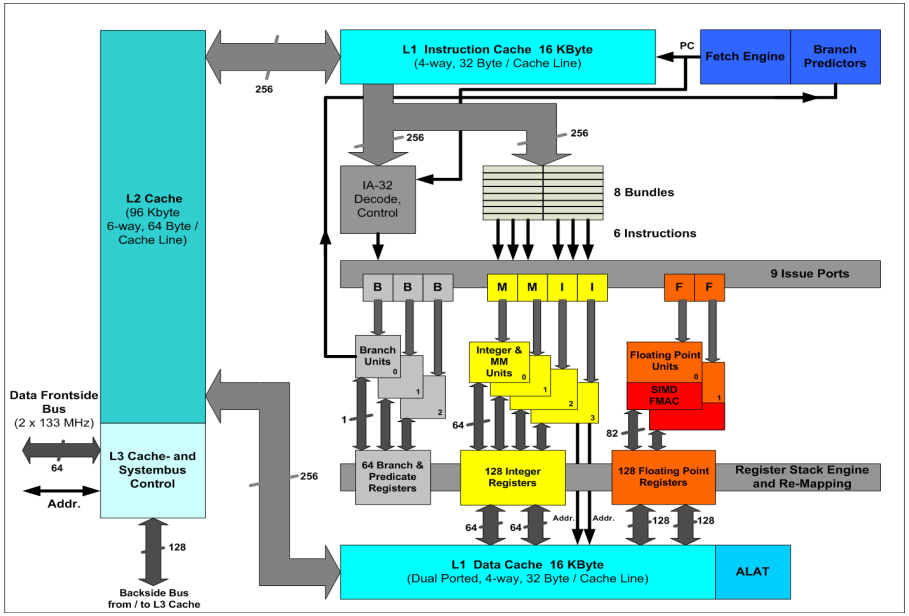
- A number of functional units that independently execute instructions in parallel.
- The compiler decides which instructions can execute in parallel
- No hazard detection needed

Mem ref 1	Mem ref 2	FP op 1	FP op 2	Int op/ branch	Clock
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1), F4	SD -8(R1), F8	ADDD F28,F26,F2			6
SD -16(R1), F12	SD -24(R1), F8				7
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#48	8
SD 0(R1),F28				BNEZ R1,LOOP	9

Itanium instruction format



Itanium architecture



Limits of VLIW

- ***Limited Instruction Level Parallelism***
  - With *n* functional units and *k* pipeline stages we need *n* x *k* independent instructions to utilize the hardware
- Memory and register bandwidth
  - With increasing number of functional units, the number of ports needed at the memory or register file must increase to prevent structural hazards
- Code size
  - Compiler scheduled pipeline “bubbles” take up space in the instruction
  - Need more aggressive loop unrolling to work well which also increases code size
- No binary code compatibility

Outline

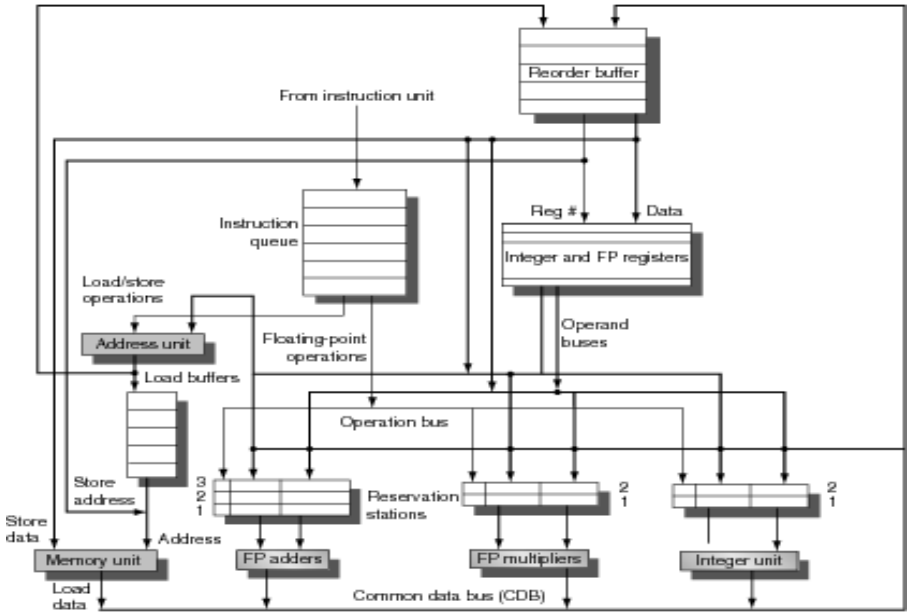
HW supported speculation

- A combination of three main ideas:
  - ***Dynamic instruction scheduling***; take advantage of ILP
  - ***Dynamic branch prediction***; allows instruction scheduling across branches
  - ***Speculative execution***; execute instructions before all control dependencies are resolved
- Hardware based speculation uses data-flow execution: ***instructions execute when their operands are available***

HW vs. SW speculation

- Advantages:
  - Dynamic runtime disambiguation of memory addresses
  - Dynamic branch prediction is often better than static which limits the performance of SW speculation
  - HW speculation can maintain a precise exception model
  - Can achieve higher performance on older code (without recompilation)
- Main disadvantage:
  - Extremely complex implementation and extensive need for hardware resources

Tomasulo extended to handle speculation



Re-order buffer - ROB

Data structure

entry	instruction type	destination	value	ready
1				
2				
...				
n				

- supports speculative execution
- instructions commit in order
- precise exceptions

Four steps of Speculative Tomasulo

- ***Issue*** – get instruction from FP Op Queue
  If reservation station ***and reorder buffer slot*** free, issue instr & send operands ***& reorder buffer nr. for destination***
- ***Execution*** – operate on operands (EX)
  If both operands ready: *execute*; if not, watch CDB for result; when both operands are in reservation station: *execute*
- ***Write result*** – complete execution
  Write on Common Data Bus to all awaiting FUs ***& reorder buffer***; mark reservation station available
- ***Commit*** – ***update register with reorder result***
  When instr. is at head of reorder buffer & result is present; update register with result (or store to memory) and remove instr. from reorder buffer;
  (handle misspeculations and precise exceptions)

Misspeculation!

Commit – ***branch prediction wrong***
When branch instr. is at head of reorder buffer & incorrect prediction:
remove all instr. from reorder buffer (flush);
restart execution at correct instruction

- Expensive $\Longrightarrow$ try to recover as early as possible
- Performance sensitive to branch prediction/speculation mechanism

Multiple issue and speculation

- Possible to extend Tomasulo with both multiple issue and speculation.
- Major issues – instruction issue and monitoring CDB
- Must be able to handle multiple commits
- Alternative to Tomasulo is to use extra physical registers for both architecturally visible registers and temporary values with register renaming

Tomasulo speculation - increased complexity

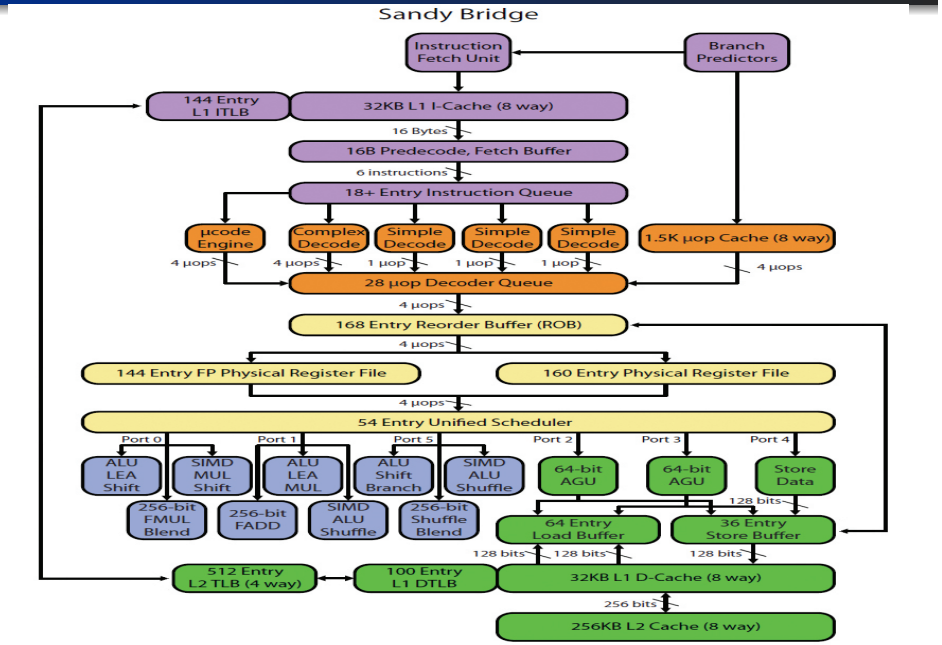
nstruction state	Wait until	Action or bookkeeping
Issue FP operation	Station r empty	`if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; RS[r].Busy ← yes; RegisterStat[rd].Q ← r;`
Load or store	Buffer r empty	`if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes;`
Load only		`RegisterStat[rt].Qi ← r;`
Store only		`if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`
Execute FP operation	(RS[r].Qj = 0) and (RS[r].Qk = 0)	Compute result: operands are in Vj and Vk
Load-store step 1	RS[r].Qj = 0 & r is head of load-store queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 complete	Read from Mem[RS[r].A]
Write Result FP operation or load	Execution complete at r & CDB available	`∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); RS[r].Busy ← no;`
Store	Execution complete at r & RS[r].Qk = 0	`Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no;`

Status	Wait until	Action or bookkeeping
Issue all instructions	Reservation station (r) and ROB (b) both available	`if (RegisterStat[rs].Busy)/*in-flight instr. writes rs*/ {h ← RegisterStat[rs].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;} else {RS[r].Qj ← h;} /* wait for instruction */ } else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;}; RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd;ROB[b].Ready ← no;`
FP operations and stores		`if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/ {h ← RegisterStat[rt].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;} else {RS[r].Qk ← h;} /* wait for instruction */ } else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;};`
FP operations		`RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd;`
Loads		`RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;`
Stores		`RS[r].A ← imm;`
Execute FP op	(RS[r].Qj == 0) and (RS[r].Qk == 0)	Compute results—operands are in Vj and Vk
Load step 1	(RS[r].Qj == 0) and there are no stores earlier in the queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 done and all stores earlier in ROB have different address	Read from Mem[RS[r].A]
Store	(RS[r].Qj == 0) and store at queue head	`ROB[h].Address ← RS[r].Vj + RS[r].A;`
Write result all but store	Execution done at r and CDB available	`b ← RS[r].Dest; RS[r].Busy ← no; ∀x(if (RS[x].Qj==b) {RS[x].Vj ← result; RS[x].Qj ← 0}); ∀x(if (RS[x].Qk==b) {RS[x].Vk ← result; RS[x].Qk ← 0}); ROB[b].Value ← result; ROB[b].Ready ← yes;`
Store	Execution done at r and (RS[r].Qk == 0)	`ROB[h].Value ← RS[r].Vk;`
Commit	Instruction is at the head of the ROB (entry h) and ROB[h].ready == yes	`d ← ROB[h].Dest; /* register dest, if exists */ if (ROB[h].Instruction==Branch) {if (branch is mispredicted) {clear ROB[h], RegisterStat; fetch branch dest;};} else if (ROB[h].Instruction==Store) {Mem[ROB[h].Destination] ← ROB[h].Value;} else /* put the result in the register destination */ {Regs[d] ← ROB[h].Value;}; ROB[h].Busy ← no; /* free up ROB entry */ /* free up dest register if no one else writing it */ if (RegisterStat[d].Reorder==h) {RegisterStat[d].Busy ← no;};`

Dynamic scheduling, speculation - summary

- tolerates unpredictable delays
- compile for one pipeline - run effectively on another
- allows speculation
  - multiple branches
  - in-order commit
  - precise exceptions
  - time, energy; recovery
- significant increase in HW complexity
- out-of-order execution, completion
- register renaming

Sandy Bridge microarchitecture



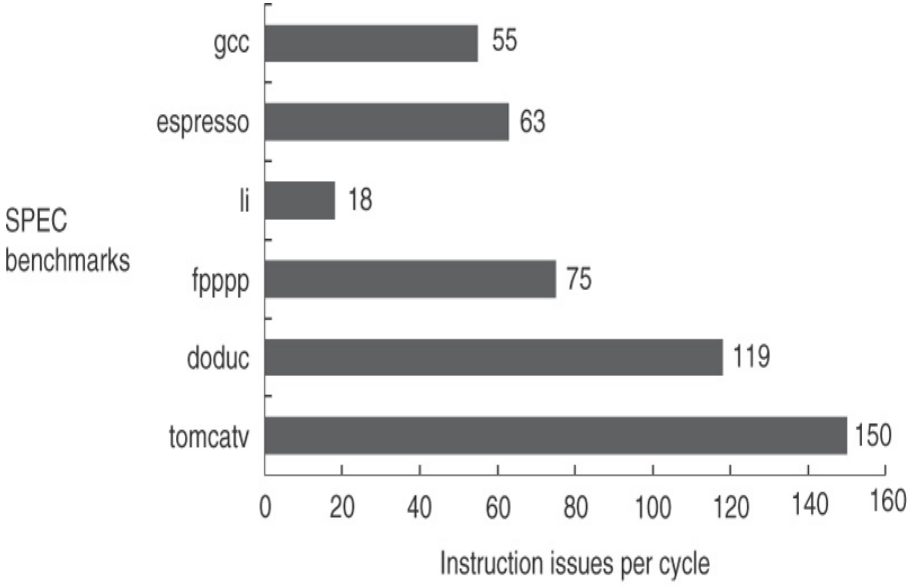
Outline

How much performance can we get by utilizing ILP?

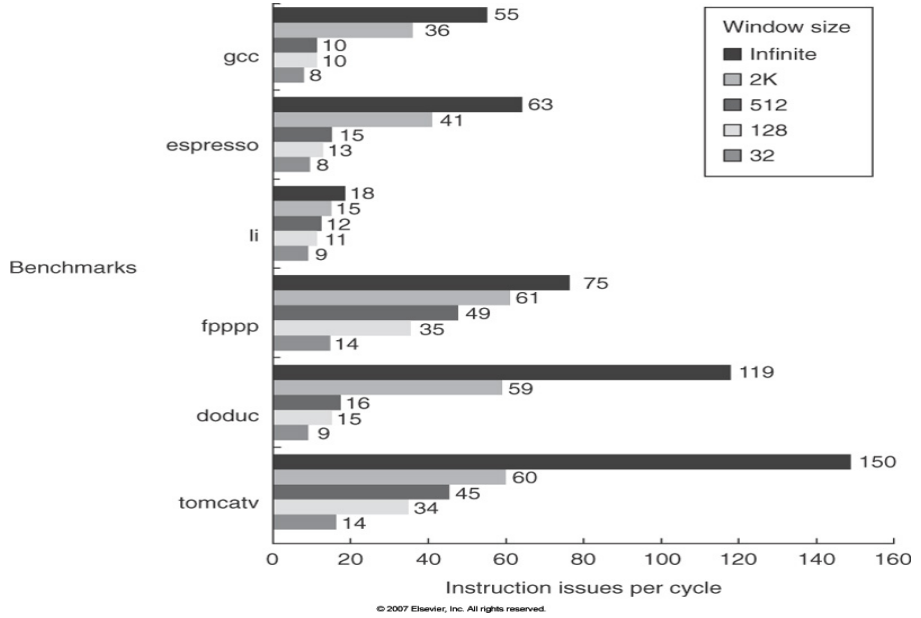
A model of an ideal processor

- Provides a base for ILP measurements
  - No structural hazards
  - *Register renaming* – infinite virtual registers and all WAW & WAR hazards avoided
  - Machine with perfect speculation
    - *Branch prediction* – perfect; no mispredictions
    - *Jump prediction* – all jumps perfectly predicted
  - Memory-address alias analysis – addresses are known & a store can be moved before a load provided addresses not equal
  - Perfect caches
- ***There are only true data dependencies left!***

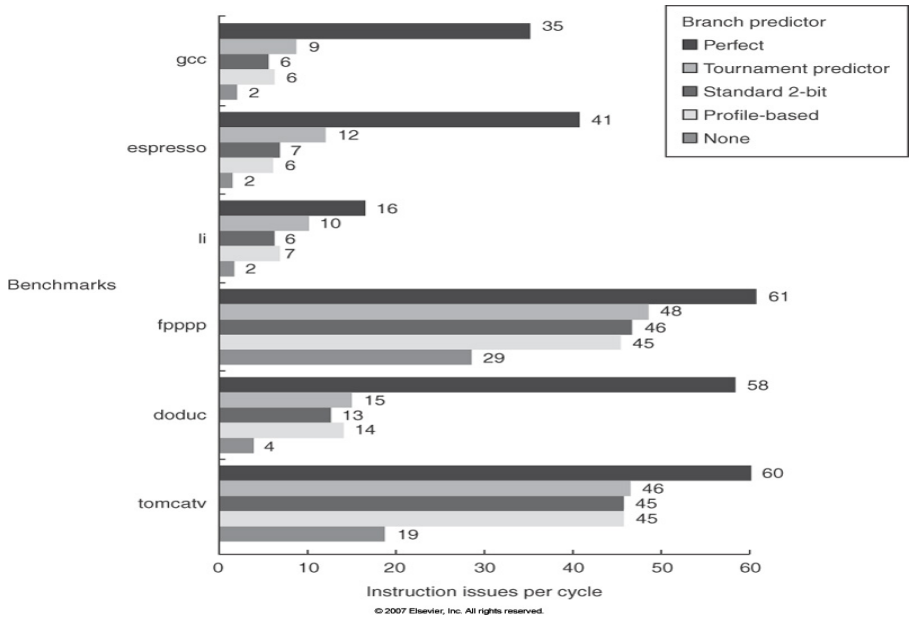
Upper Limit to ILP



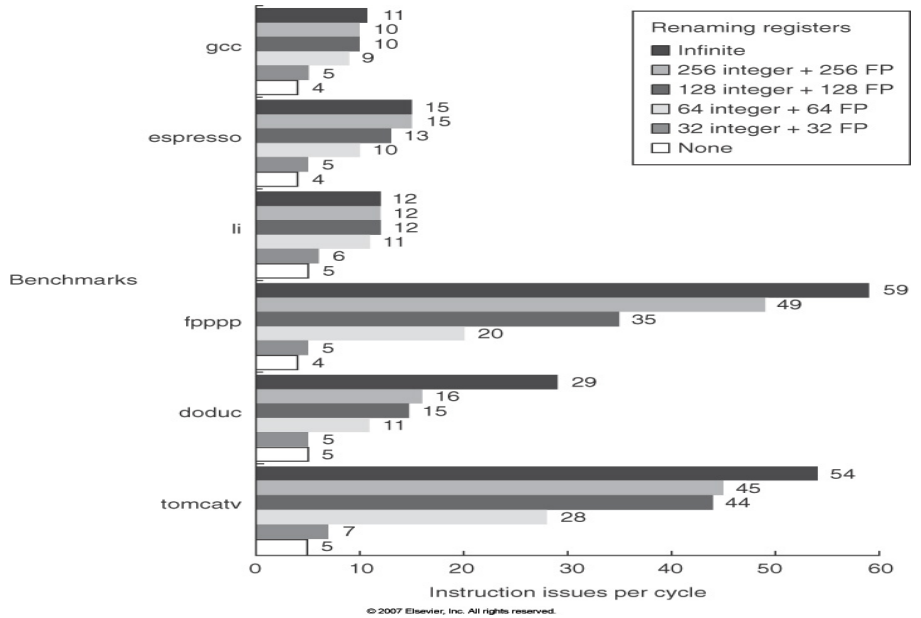
Impact window size



More realistic HW: Branch impact



More realistic HW: Register impact



Summary

Software (*compiler*) tricks:

- Loop unrolling
- Static instruction scheduling (with *register renaming*)
- ... and more

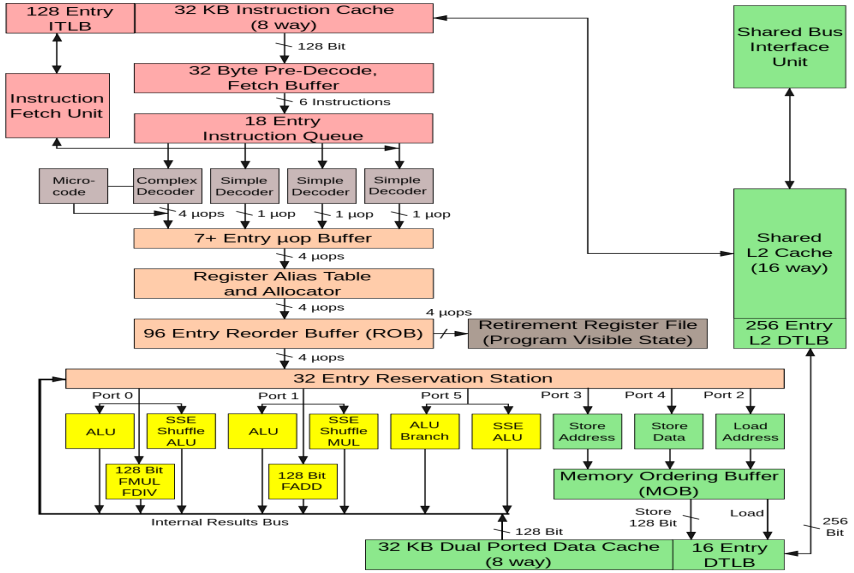
Hardware tricks:

- Dynamic instruction scheduling
- Dynamic branch prediction
- Multiple issue – Superscalar, VLIW
- Speculative execution
- ... and more

Outline

AMD Phenom CPU

Intel Core2



Intel Core 2 Architecture

Intel Core2 chip (Nehalem)