Concepts Introduced in Chapter 3

- introduction
- compiler techniques for ILP
- advanced branch prediction
- dynamic scheduling
- speculation
- multiple issue
- limits of ILP
- multithreading

Instruction Level Parallelism (ILP)

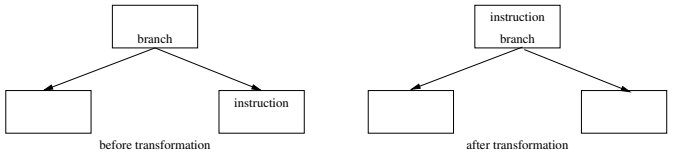
- Pipelining is the overlapping of different portions of instruction execution.
- Instruction level parallelism is the parallel execution of a sequence of instructions associated with a single thread of execution.
- scheduling approaches to exploit ILP
  - static (compiler)
  - dynamic (hardware)

Data Dependences

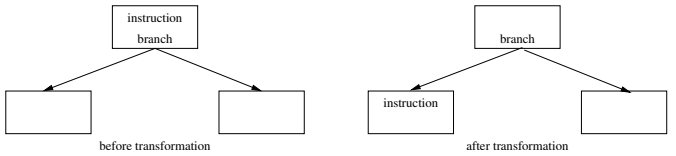
- Instructions must be independent to be executed in parallel.
- types of data dependences
  - True dependences can lead to RAW hazards.
  - name dependences
    - Antidependences can lead to WAR hazards.
    - Output dependences can lead to WAW hazards.

Control Dependences

- An instruction is control dependent on a branch instruction if the instruction will only be executed when the branch has a specific result.
- An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is not controlled by the branch.



- An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.



Static Scheduling

- A statically scheduled processor relies on an in-order instruction pipeline.
- If a hazard is detected that cannot be resolved on a statically scheduled processor, then the instruction is stalled in the ID stage and no new instructions are fetched or issued until the dependence is cleared.
- Compiler techniques are used to statically schedule the instructions to avoid or minimize stalls.
  - Increase the distance between dependent instructions.
  - Perform other transformations (e.g. register renaming).

Latencies of FP Operations Used in This Chapter

- The latency here indicates the number of intervening independent instructions that are needed to avoid a stall.

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

```
FPALUOP  IF   ID   FEX   FEX   FEX   FEX   FWB
FPALUOP       IF   ID    stall stall stall FEX   FEX   FEX   FEX   FWB

FPALUOP  IF   ID   FEX   FEX   FEX   FEX   FWB
STD           IF   ID    stall stall EX    MEM

LDD      IF   ID   EX    MEM   FWB
FPALUOP       IF   ID    stall FEX   FEX   FEX   FEX   FWB

LDD      IF   ID   EX    MEM   FWB
STD           IF   ID    EX    MEM
```


Example Loop

- Assume the following source code is compiled into the following RISC-V assembly code.

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

=>

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


Instruction Scheduling

```
loop: fld    f0,0(x1)    # original assembly code
      fadd.d f4,f0,f2
      fsd    f4,0(x1)
      addi   x1,x1,-8
      bne    x1,x2,loop

loop: fld    f0,0(x1)    # original code with stalls
      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)    # scheduled code
      addi   x1,x1,-8
      fadd.d f4,f0,f2
      stall
      stall
      fsd    f4,8(x1)
      bne    x1,x2,loop
```

Loop Unrolling

```
for (i = 0; i < n; i++)
   a[i] = a[i] + x;
=>
for (i = 0; i < n-3; i++) {
   a[i] = a[i] + x; i++;
   a[i] = a[i] + x; i++;
   a[i] = a[i] + x; i++;
   a[i] = a[i] + x;
}
for (; i < n; i++)
   a[i] = a[i] + x;
=>
for (i = 0; i < n-3; i += 4) {
   a[i] = a[i] + x;
   a[i+1] = a[i+1] + x;
   a[i+2] = a[i+2] + x;
   a[i+3] = a[i+3] + x;
}
for (; i < n; i++)
   a[i] = a[i] + x;
```


Loop Unrolling at the Assembly Level

unrolled loop		after moving addi insts	
loop:		loop:	
fld	f0,0(x1)	fld	f0,0(x1)
fadd.d	f4,f0,f2	fadd.d	f4,f0,f2
fsd	f4,0(x1)	fsd	f4,0(x1)
addi	x1,x1,-8	fld	f6,-8(x1)
fld	f6,0(x1)	fadd.d	f8,f6,f2
fadd.d	f8,f6,f2	fsd	f8,-8(x1)
fsd	f8,0(x1)	fld	f10,-16(x1)
addi	x1,x1,-8	fadd.d	f12,f10,f2
fld	f10,0(x1)	fsd	f12,-16(x1)
fadd.d	f12,f10,f2	fld	f14,-24(x1)
fsd	f12,0(x1)	fadd.d	f16,f14,f2
addi	x1,x1,-8	fsd	f16,-24(x1)
fld	f14,0(x1)	addi	x1,x1,-8
fadd.d	f16,f14,f2	addi	x1,x1,-8
fsd	f16,0(x1)	addi	x1,x1,-8
addi	x1,x1,-8	addi	x1,x1,-8
bne	x1,x2,loop	bne	x1,x2,loop

Loop Unrolling and Scheduling

after coalesing adds		after showing stalls			
loop:		loop:			
fld	f0,0(x1)	fld	f0,0(x1)	fadd.d	f12,f10,f2
fadd.d	f4,f0,f2	stall		stall	
fsd	f4,0(x1)	fadd.d	f4,f0,f2	stall	
fld	f6,-8(x1)	stall		fsd	f12,-16(x1)
fadd.d	f8,f6,f2	stall		fld	f14,-24(x1)
fsd	f8,-8(x1)	fsd	f4,0(x1)	stall	
fld	f10,-16(x1)	fld	f6,-8(x1)	fadd.d	f16,f14,f2
fadd.d	f12,f10,f2	stall		stall	
fsd	f12,-16(x1)	fadd.d	f8,f6,f2	stall	
fld	f14,-24(x1)	stall		fsd	f16,-24(x1)
fadd.d	f16,f14,f2	stall		addi	x1,x1,-32
fsd	f16,-24(x1)	fsd	f8,-8(x1)	bne	x1,x2,loop
addi	x1,x1,-32	fld	f10,-16(x1)		
bne	x1,x2,loop	stall			

Loop Unrolling and Scheduling (cont.)

after coalesing adds		after scheduling	
loop:		loop:	
fld	f0,0(x1)	fld	f0,0(x1)
fadd.d	f4,f0,f2	fld	f6,-8(x1)
fsd	f4,0(x1)	fld	f10,-16(x1)
fld	f6,-8(x1)	fld	f14,-24(x1)
fadd.d	f8,f6,f2	fadd.d	f4,f0,f2
fsd	f8,-8(x1)	fadd.d	f8,f6,f2
fld	f10,-16(x1)	fadd.d	f12,f10,f2
fadd.d	f12,f10,f2	fadd.d	f16,f14,f2
fsd	f12,-16(x1)	fsd	f4,0(x1)
fld	f14,-24(x1)	fsd	f8,-8(x1)
fadd.d	f16,f14,f2	fsd	f12,-16(x1)
fsd	f16,-24(x1)	fsd	f16,-24(x1)
addi	x1,x1,-32	addi	x1,x1,-32
bne	x1,x2,loop	bne	x1,x2,loop

Dependences and Instruction Scheduling

- Dependences restrict the movement of instructions and the order in which results must be calculated.
- data (true) dependences
  - True dependences cannot be avoided.
- name (false) dependences
  - Anti and output dependences can sometimes be avoided by register renaming.
- control dependences
  - Control dependences can sometimes be avoided through branch elimination techniques or speculation.

True Dependences

- Data (true) dependences occur when the result of one instruction is used by another and can lead to RAW hazards.
- The following sequence of instructions cannot be reordered.

```
fld    f0,0(x1)
fadd.d f4,f0,f2
fsd    f4,0(x1)
```


Name Dependences

- Name dependences can sometimes be avoided by using additional registers (register renaming).

before renaming

```
loop:
  fld    f0,0(x1)
  fadd.d f4,f0,f2
  fsd    f4,0(x1)
  fld    f0,-8(x1)
  fadd.d f4,f0,f2
  fsd    f4,-8(x1)
  fld    f0,-16(x1)
  fadd.d f4,f0,f2
  fsd    f4,-16(x1)
  fld    f0,-24(x1)
  fadd.d f4,f0,f2
  fsd    f4,-24(x1)
  addi   x1,x1,-32
  bne    x1,x2,loop
```

after renaming

```
loop:
  fld    f0,0(x1)
  fadd.d f4,f0,f2
  fsd    f4,0(x1)
  fld    f6,-8(x1)
  fadd.d f8,f6,f2
  fsd    f8,-8(x1)
  fld    f10,-16(x1)
  fadd.d f12,f10,f2
  fsd    f12,-16(x1)
  fld    f14,-24(x1)
  fadd.d f16,f14,f2
  fsd    f16,-24(x1)
  addi   x1,x1,-32
  bne    x1,x2,loop
```


Control Dependences

before branch removal

```
loop:  fld    f0,0(x1)
       fadd.d f4,f0,f2
       fsd    f4,0(x1)
       addi   x1,x1,-8
       beq    x1,x2,exit
       fld    f6,0(x1)
       fadd.d f8,f6,f2
       fsd    f8,0(x1)
       addi   x1,x1,-8
       beq    x1,x2,exit
       fld    f10,0(x1)
       fadd.d f12,f10,f2
       fsd    f12,0(x1)
       addi   x1,x1,-8
       beq    x1,x2,exit
       fld    f14,0(x1)
       fadd.d f16,f14,f2
       fsd    f16,0(x1)
       addi   x1,x1,-8
       bne    x1,x2,loop
exit:
```

after branch removal

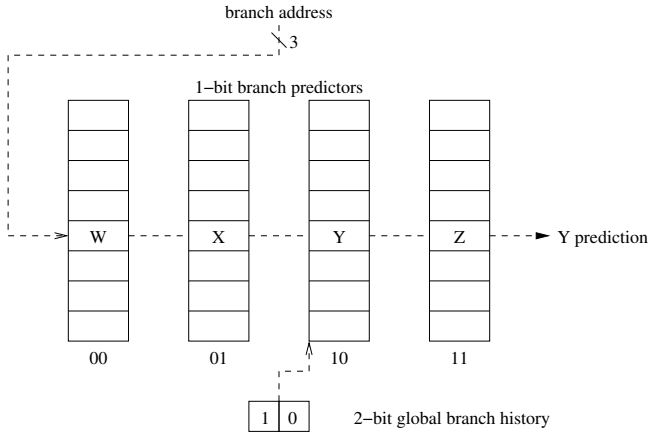
```
loop:  fld    f0,0(x1)
       fadd.d f4,f0,f2
       fsd    f4,0(x1)
       addi   x1,x1,-8
       fld    f6,0(x1)
       fadd.d f8,f6,f2
       fsd    f8,0(x1)
       addi   x1,x1,-8
       fld    f10,0(x1)
       fadd.d f12,f10,f2
       fsd    f12,0(x1)
       addi   x1,x1,-8
       fld    f14,0(x1)
       fadd.d f16,f14,f2
       fsd    f16,0(x1)
       addi   x1,x1,-8
       bne    x1,x2,loop
```

Correlating Branch Prediction Buffers (BPBs)

- An (m,n) predictor uses the behavior of the last m branches executed to choose from 2^m branch predictors, each of which is n-bits.
- A (0,1) predictor would be the conventional 1-bit predictor.
- A (0,2) predictor would be the conventional 2-bit predictor. Usually machines never have more than 2 bits for each predictor.
- A correlating predictor records the most recent m branch results using an m-bit shift register, where each bit indicates if the corresponding branch was taken or not taken.

Correlating Branch Predictor (BP) Hardware Example

- Below is an example of a (2,1) branch-prediction buffer. It uses a 2-bit global history to choose from among four 1-bit predictors for each branch address with a total of 32 entries.



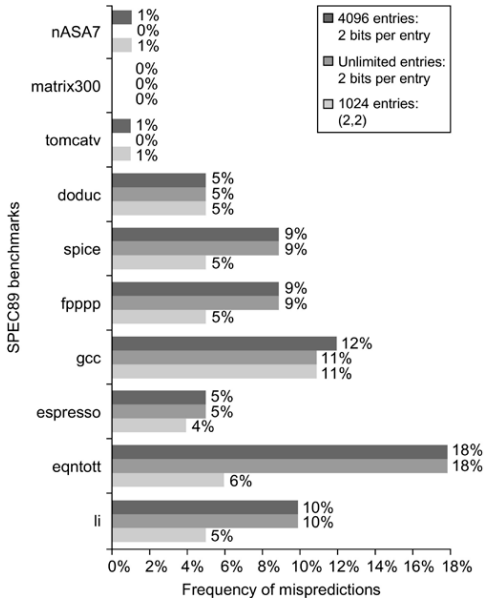
Correlating Branch Predictor Example

- Consider the following example with a (2,1) predictor.

```
source code:
   for (i = 0; i < 100; i++)
      if (i & 1) /* i is odd */
         ...
assembly code level:
   loop: if ((i & 1) == 0) goto end;
         ...
    end: i++;
         if (i < 100) goto loop;
```

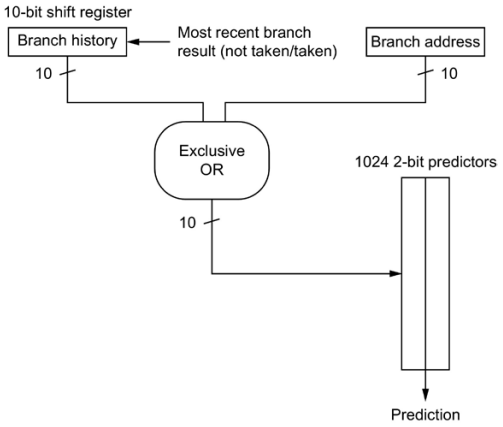
branch	branch history			
	00 (initial state)	**01 (if NT/loop T)** 01 (loop NT/if T)	**10 (loop T/if NT)** 10 (if T/loop NT)	**11 (both T)**
if				
loop				

Comparison of 2-Bit Predictors



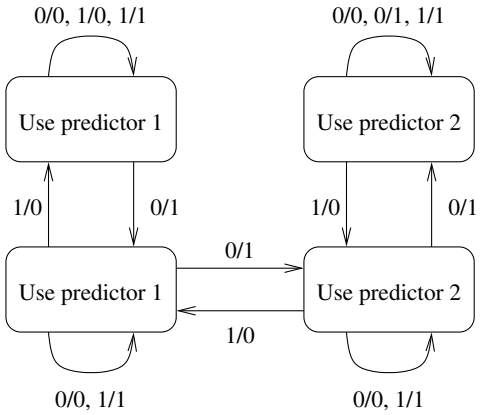
Gshare Predictor

- The best known correlating predictor is the gshare predictor, which uses an exclusive OR of the branch history and the lower portion of the branch address to determine the index.



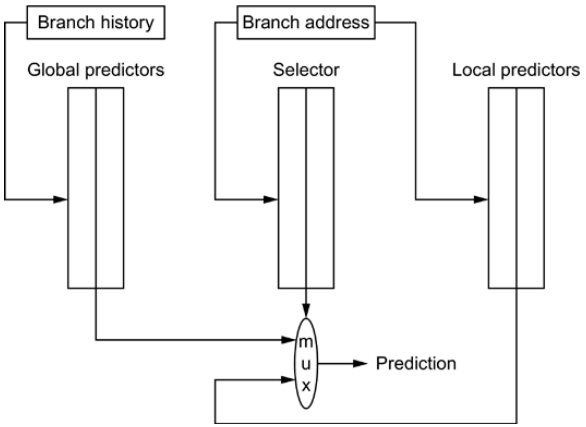
Tournament Branch Prediction Buffers

- Uses a local predictor (uses no global history) and a global predictor (uses global history).
- Uses a selector, similar to the 2-bit predictor, to decide whether to use the result of the local or global predictor.

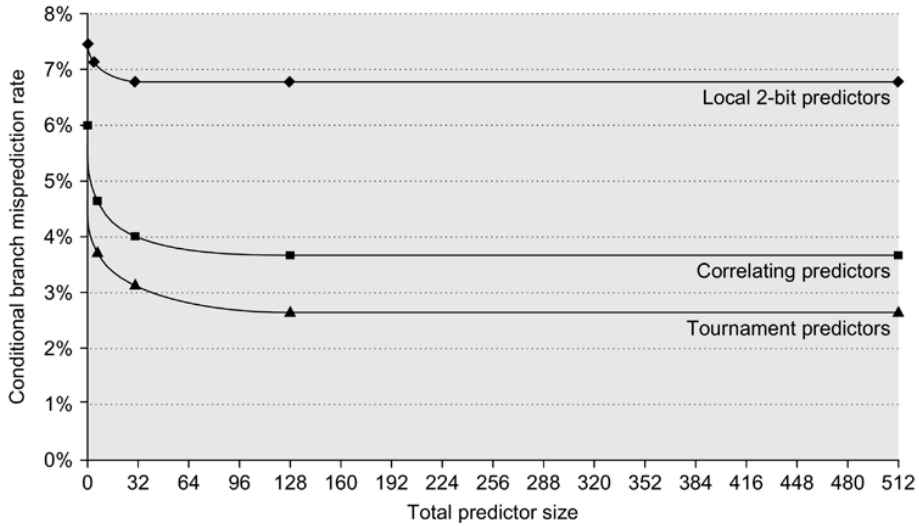


Tournament Branch Predictor Example

- Uses branch history to index into global (correlating) predictor.
- Uses branch address to index into selector and local (2-bit) predictors.
- The selector drives a multiplexor to choose between the local and global predictions.

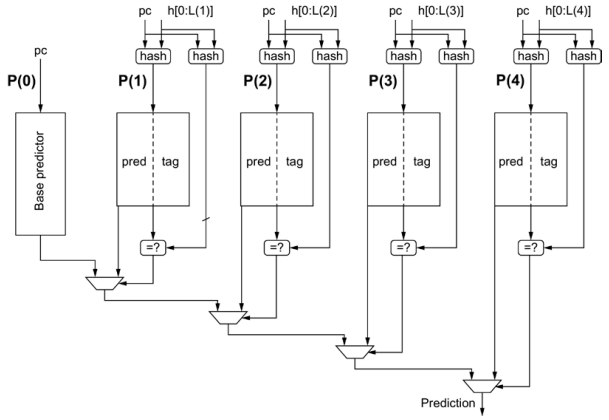


Misprediction Rate for Three Different Predictors



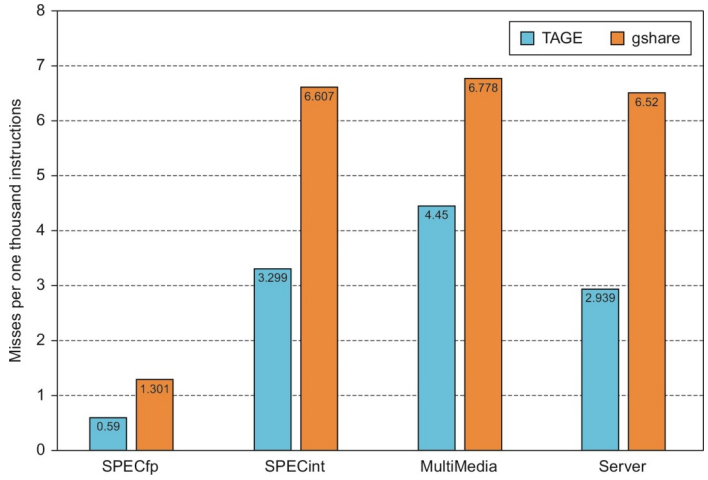
Tagged Hybrid Predictors

- The current best performing branch prediction approaches utilize a series of global predictors with different length global histories and small tags to match the hash of the branch address and global history.
- Chooses the prediction with the longest history and a matching tag.



Comparison of TAGE versus Gshare Prediction Schemes

- Both prediction schemes use the same number of bits.
- Can see that the TAGE predictors have lower miss rates.

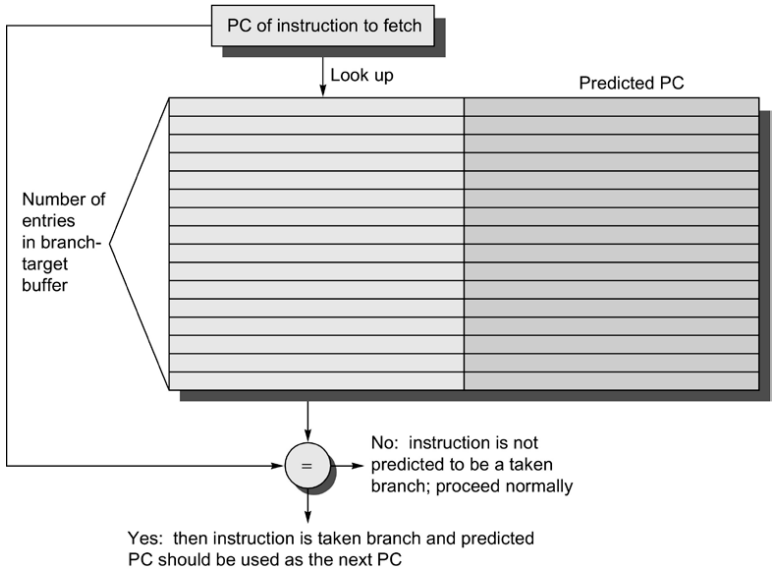


Branch Target Buffer (BTB)

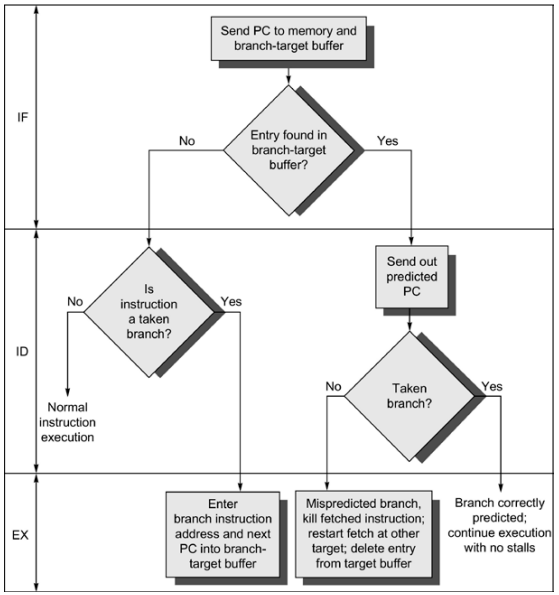
- Stores the actual branch target address.
- Checked during the IF stage to allow earlier resolution.
- A tag is needed to make sure the instruction is the correct branch.
- Not put into the buffer until the branch is taken.
- The *next* instruction below can either be the fall-through or the target instruction as both the BTB and the BP are accessed during the IF stage.

```
branch IF  ID  EX MEM WB
next       IF  ID EX  MEM WB
```


Branch Target Buffer Example



Steps in Handling an Instruction with a BTB



BTB Penalty Cycles

- Here we assume the branch is predicted taken if it is in the buffer and that branches are resolved at the end of the ID stage.
- Only taken branches are stored in the buffer.

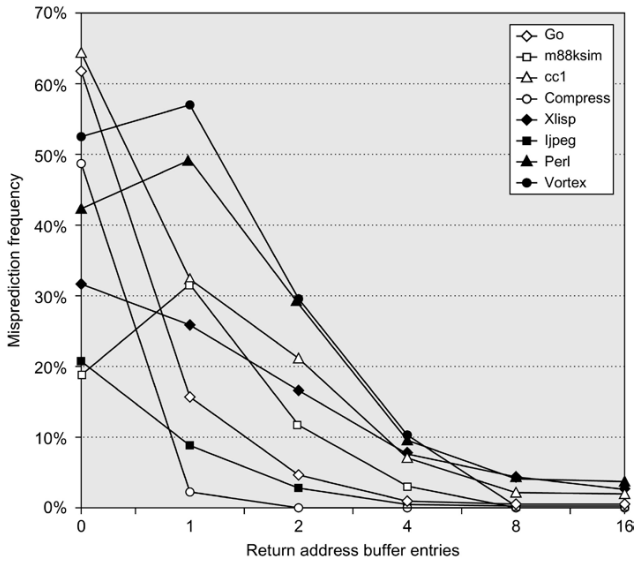
Instruction in buffer	Prediction	Actual branch	Penalty cycles
Yes	Taken	Taken	0
Yes	Taken	Not taken	2
No		Taken	2
No		Not taken	0

Return Prediction

- A circular buffer of return addresses called the return address stack (RAS) is used for predicting return addresses.
- Pushes the return address on the RAS at each call.
- Pops the return address from the RAS at each return.
- Can either add a bit to the BTB or keep a predecode bit in the L1 IC to recognize that the current instruction is a return during the instruction fetch stage so the address on the top of the RAS should be used.
- Works well as long as the calling depth does not exceed the number of entries in the RAS, a context switch does not occur, and a callee always returns to the caller (no *longjmp* operations).
- The pipeline has to check if the return address popped from the RAS matches the address in the return address register.

Prediction Accuracy for a RAS

- A buffer of 0 indicates no RAS is used and a regular BP is used.



Dynamic Scheduling

- The hardware is designed to rearrange the order in which instructions are executed to reduce pipeline stalls. Can be used to avoid WAW and WAR hazards.
- Advantages
  - does not require recompilation to be exploited
  - can handle memory dependences that are unknown at compile time
  - can execute other instructions when encountering cache misses
- Disadvantages
  - more complicated hardware
  - more energy usage
  - probably a longer cycle time

Out-of-Order Execution

- Dynamic scheduling allows out-of-order (OoO) instruction execution.
- In the example below, the `fsub.d` can execute while the `fadd.d` is stalled as the `fsub.d` does not depend on the `fdiv.d` or the `fadd.d`.

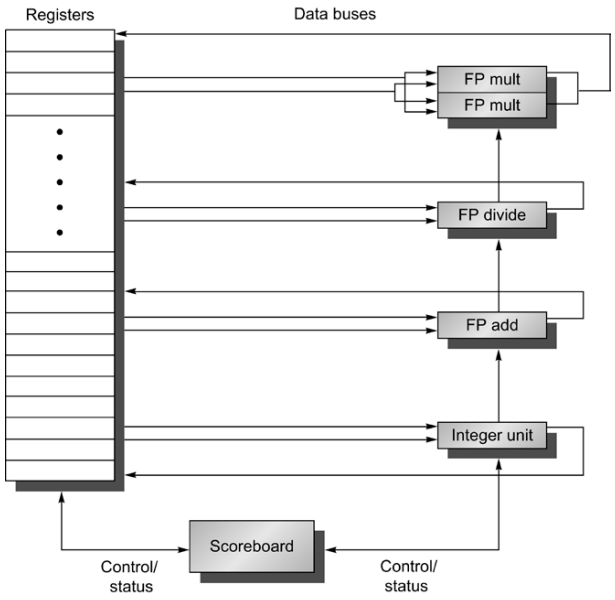
```
fdiv.d f0,f2,f4
fadd.d f10,f0,f8
fsub.d f12,f8,f14
```

- The ID stage is split into two parts.
  - Issue: Decode instructions and check for structural hazards.
  - Read Operands: Wait until no RAW hazards.
- OoO execution can result in OoO completion, which complicates exception handling.

Scoreboarding Approach

- Scoreboarding was used to allow OoO execution when there are no structural hazards and no data dependences.
- Was used in the CDC 6600.
- Can be achieved by using multiple functional units.

Basic Structure of a Scoreboard



Scoreboarding Steps

- Issue - Get an instruction from the instruction queue. Issue it if the scoreboard indicates that there is an available functional unit and no other active instruction has the same destination register. Otherwise stall the issue stage. Checks for structural and WAW hazards.
- Read operands - When the source operands for an active instruction in a functional unit are available, read the operands from the register file and indicate to the scoreboard that execution may start. Checks for RAW hazards.
- Execute - Start execution when operands have been read. Notify the scoreboard when execution is completed.
- Write result - If there is a preceding active instruction that has not read one of its operands that is the same register as the destination of the completing instruction, then stall the completing instruction. Otherwise store the result to the register file. Checks for WAR hazards.

Components of a Scoreboard

		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 Just before the MUL.D Writes Result

		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 Just before the DIV.D Writes Result

		Instruction status			
Instruction		Issue	Read operands	Execution complete	Write result
L.D	F6,34(R2)d	√	√	√	√
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	√	√	√	√

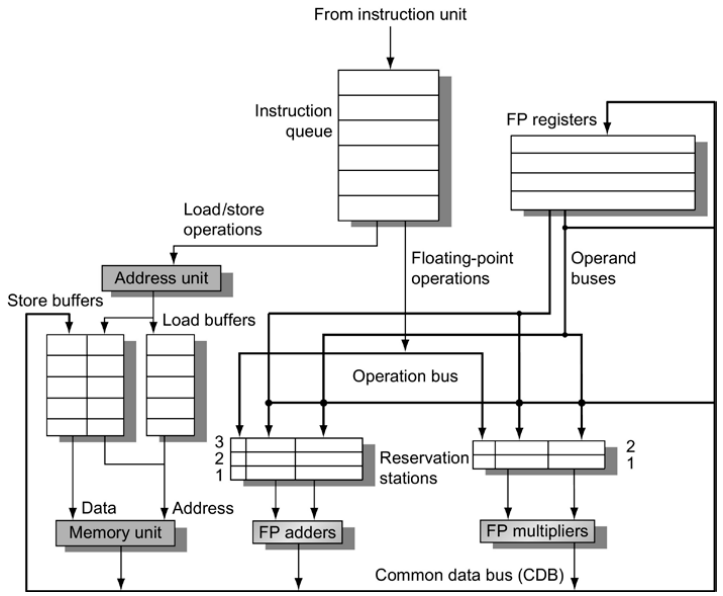
	Functional unit status								
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			No	No
Mult2	No								
Add	Yes	Add	F6	F8	F2			No	No
Divide	Yes	Div	F10	F0	F6			No	Yes

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

Tomasulo Approach

- Is used to avoid WAR and WAW hazards.
- Data values are often directly obtained from buffers or functional units via a common data bus (CDB).
- Uses dynamic register renaming, which is accomplished by reservation stations. As instructions are issued, the register specifiers for pending operands are renamed to be the names of the reservation stations or load buffer entries.
- Uses dynamic scheduling (out-of-order execution).
- Uses dynamic memory disambiguation.

Basic Structure of FP Unit Using Tomasulo's Algorithm



Tomasulo Algorithm Steps

- Issue - Get an instruction from the instruction queue. Issue it if there is an empty reservation station and send the operands to this reservation station if they are available in registers. Checks for structural hazards.
- Execute - Monitor the CDB waiting for the operands to become available. When both operands are available, execute the operation. Checks for RAW hazards.
- Write result - Write the result on the CDB when it is available. Indicate which register will receive that value. The value will also be forwarded to and stored in the appropriate reservation stations waiting for the result.

All Instructions Issued and 1st Load Has Written Its Result

	Instruction status		
Instruction	Issue	Execute	Write result
`fld f6,32(x2)`	√	√	√
`fld f2,44(x3)`	√	√	
`fmul.d f0,f2,f4`	√		
`fsub.d f8,f2,f6`	√		
`fdiv.d f0,f0,f6`	√		
`fadd.d f6,f8,f2`	√		

	Reservation stations							
Name	Busy	Op	Vj	Vk	Qj	Qk	A	
Load1	No							
Load2	Yes	Load						`44 + Regs[x3]`
Add1	Yes	SUB		`Mem[32 + Regs[x2]]`	Load2			
Add2	Yes	ADD			Add1	Load2		
Add3	No							
Mult1	Yes	MUL		`Regs[f4]`	Load2			
Mult2	Yes	DIV		`Mem[32 + Regs[x2]]`	Mult1			

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	...	f30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Only Multiply and Divide Have Not Finished

		Instruction status		
Instruction		Issue	Execute	Write result
fld	f6,32(x2)	√	√	√
fld	f2,44(x3)	√	√	√
fmul.d	f0,f2,f4	√	√	
fsub.d	f8,f2,f6	√	√	√
fdiv.d	f0,f0,f6	√		
fadd.d	f6,f8,f2	√	√	√

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	No						
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL	Mem[44 + Regs[x3]]	Regs[f4]			
Mult2	Yes	DIV		Mem[32 + Regs[x2]]	Mult1		

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	...	f30
Qi	Mult1					Mult2			

Tomasulo Algorithm Example 1

- execution assumptions
  - one cycle for integer ALU operations
  - two cycles for a load (one for address calculation and one for data cache access)
  - two cycles for fadd.d, 4 cycles for fmul.d, 10 cycles for fdiv.d

Instruction	Issues at	Execute Start	Execute End	Memory Access at	Write CDB at
fld f6,32(x2)	1	2	2	3	4
fld f2,44(x3)					
fmul.d f0,f2,f4					
fsub.d f8,f6,f2					
fdiv.d f10,f0,f6					
fadd.d f6,f8,f2					

Tomasulo Algorithm Example 2

- execution assumptions
  - one cycle for integer ALU operations and branches
  - two cycles for loads and stores
  - four cycles for fmul.d

Instruction	Issues at	Execute Start	Execute End	Memory Access at	Write CDB at
fld f0,0(x1)	1	2	2	3	4
fmul.d f4,f0,f2					
fsd f4,0(x1)					
addi x1,x1,-8					
bne x1,x2,Loop					
fld f0,0(x1)					
fmul.d f4,f0,f2					
fsd f4,0(x1)					
addi x1,x1,-8					
bne x1,x2,Loop					

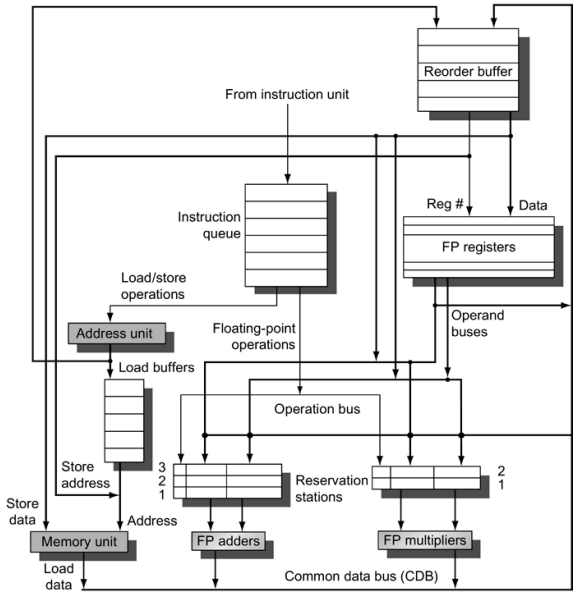
Hardware-Based Speculation

- Combines three key ideas.
  - dynamic branch prediction
  - execution of instructions before control dependences are resolved
  - dynamic scheduling (out-of-order execution)
- We will see that instructions on such a machine:
  - Issue in order.
  - Can execute out of order.
  - Update the state of the machine (commit) in order.

Speculative Tomasulo Approach

- Allows speculative execution with dynamic scheduling based on Tomasulo's algorithm.
- Like Tomasulo's algorithm, it allows instructions to execute out of order.
- Unlike Tomasulo's algorithm, it forces the instructions to commit (update registers or memory) in order. The advantages are:
  - Supports speculative execution.
  - Precise exceptions are supported.

Basic Structure of FP Unit Using Speculative Tomasulo



Speculative Tomasulo Steps

- Issue - Get an instruction from the instruction queue. Issue it if there is an empty reservation station and an empty slot in the reorder buffer and mark both as busy. Checks for structural hazards.
- Execute - Monitor the CDB waiting for the operands to become available. When both operands are available, execute the operation. Checks for RAW hazards.
- Write result - Write the result on the CDB when it is available with the appropriate tag (reorder buffer slot #) that was stored in the reservation station. Mark the reservation station as available. The result will also be sent to the reorder buffer.
- Commit - When the result of the instruction at the head of the reorder buffer is available, update the state (memory or a register) and remove the instruction from the reorder buffer. If a branch is at the head and it is found that it was incorrectly predicted, then flush the instructions that entered after the branch out of the buffer.

Multiply Is Ready to Commit

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

Speculative Tomasulo Algorithm Example 1

- execution assumptions
  - one cycle for integer ALU operations
  - two cycles for a load and stores (one for address calculation and one for data cache access)
  - two cycles for fadd.d, 4 cycles for fmul.d, 10 cycles for fdiv.d

Instruction	Issues at	Executes at	Memory Access at	Write CDB at	Commits at
`fld f6,32(x2)`	1	2-2	3	4	5
`fld f2,44(x3)`					
`fmul.d f0,f2,f4`					
`fsub.d f8,f6,f2`					
`fdiv.d f10,f0,f6`					
`fadd.d f6,f8,f2`					

Only the Load and Multiply Have Committed

Reorder buffer

Entry	Busy	Instruction	State	Destination	Value
1	No	`fld f0,0(x1)`	Commit	`f0`	`Mem[0 + Regs[x1]]`
2	No	`fmul.d f4,f0,f2`	Commit	`f4`	#1 × `Regs[f2]`
3	Yes	`fsd f4,0(x1)`	Write result	`0 + Regs[x1]`	#2
4	Yes	`addi x1,x1,−8`	Write result	`x1`	`Regs[x1] – 8`
5	Yes	`bne x1,x2,Loop`	Write result		
6	Yes	`fld f0,0(x1)`	Write result	`f0`	`Mem[#4]`
7	Yes	`fmul.d f4,f0,f2`	Write result	`f4`	#6 × `Regs[f2]`
8	Yes	`fsd f4,0(x1)`	Write result	0 + #4	#7
9	Yes	`addi x1,x1,−8`	Write result	`x1`	#4 – 8
10	Yes	`bne x1,x2,Loop`	Write result		

FP register status

Field	f0	f1	f2	f3	f4	F5	f6	F7	f8
Reorder #	6								
Busy	Yes	No	No	No	Yes	No	No	…	No

Speculative Tomasulo Algorithm Example 2

- execution assumptions
  - one cycle for integer ALU operations and branches
  - two cycles for a load and stores (one for address calculation and one for data cache access)
  - four cycles for fmul.d

Instruction	Issues at	Executes at	Memory Access at	Write CDB at	Commits at
`fld f0,0(x1)`	1	2-2	3	4	5
`fmul.d f4,f0,f2`					
`fsd f4,0(x1)`					
`addi x1,x1,-8`					
`bne x1,x2,Loop`					
`fld f0,0(x1)`					
`fmul.d f4,f0,f2`					
`fsd f4,0(x1)`					
`addi x1,x1,-8`					
`bne x1,x2,Loop`					

Dynamic Scheduling Techniques Summary

- Scoreboarding was introduced in the CDC 6600 and supported out-of-order execution.
- Tomasulo's algorithm was introduced in the IBM 360/91 and also used register renaming to support avoiding WAR and WAW hazards.
- Speculative Tomasulo's algorithm was used in the PowerPC 620, MIPS 10000, Intel Pentium Pro, etc. and supported speculation and precise exceptions.

Hardware versus Software Speculation

- hardware speculation advantages
  - better memory disambiguation
  - better branch prediction
  - simpler compiler
  - binary code compatibility
- hardware speculation disadvantages
  - more complex design
  - more space on chip
  - may slow the processor clock cycle
  - requires more energy usage

How Much to Speculate?

- Speculation is not free due to branch mispredictions.
  - Often causes more energy usage due to unnecessarily executed instructions and the cost of recovery.
  - Can potentially result in a performance degradation, so only low-cost exceptional events (e.g. L1 DC miss) are allowed for speculative instructions.
- Some additional complexity when speculating across multiple branches, but is needed for multi-issue processors.
  - More complicated recovery after branch mispredictions.
  - How to make predictions on multiple branches in the same cycle.

Value Prediction

- Value prediction attempts to predict the value that will be produced by an instruction.
- Value prediction is another form of speculation.
- Much of the research on value prediction has been on loads.
  - Determine which loads are predictable by using profile data and mark these loads.
  - Index into table using address of instruction to verify that previous load result for the instruction is in the table when the tag matches and speculate using last value loaded.
- Must be a method for recovery when the prediction is incorrect.

Issuing Multiple Instructions in Parallel

- Issuing multiple instructions in parallel requires simultaneous actions to be performed for multiple instructions.
  - fetching
  - decoding
  - executing
  - accessing memory
  - writing results
- multiple issue processors
  - statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - dynamically scheduled superscalar processors

Statically Scheduled Superscalar Approach

- Issue multiple instructions that can be quickly detected as independent.
- Typically, statically scheduled superscalar machines can simultaneously issue at most 2 to 4 instructions.
- If one instruction in the set has some dependency, then only the preceding instructions will be issued.
- More likely to have hazards causing stalls.
- No increase in code size and less work for the compiler.
- Programs compiled for nonsuperscalar (single issue) machines will still execute and have benefits.

VLIW Approach

- VLIW stands for Very Long Instruction Word.
- Can issue many instructions in parallel.
- The compiler has the responsibility to package multiple independent instructions together.
- Simplifies issuing hardware.
- With many functional units, this requires complex compiler scheduling support.
  - loop unrolling
  - software pipelining
  - superblock scheduling
  - if conversion with predication support

VLIW Example

- Assume the compiler unrolled the loop by a factor of 7.

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

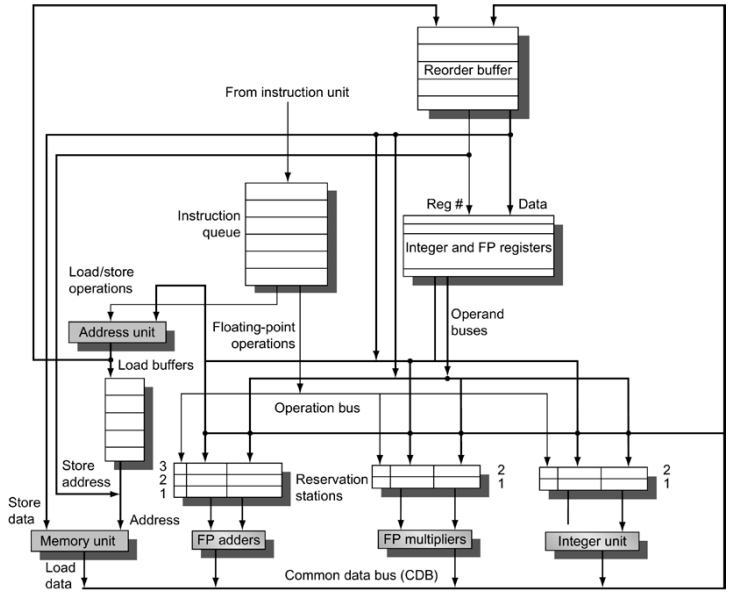
VLIW Challenges

- Performing multiple loads and stores in the same cycle.
- There could be multiple accesses to the register file in the same cycle.
- The clock cycle may have to be lengthened.
- May significantly increase code size due to inserting noop instructions and compilation techniques (e.g. loop unrolling and/or software pipelining) to extract more parallelism.
- Any functional unit stall will cause the entire processor to stall. This includes cache misses.
- Different implementations (number of functional units) results in executables that are not binary compatible.

Dynamically Scheduled Superscalar Approach

- Simultaneously fetch multiple instructions.
- Issue multiple instructions in the same cycle.
- Access multiple memory references in the same cycle.
- Write multiple results on the CDB in the same cycle.
- Commit multiple instructions in the same cycle.

Basic Organization of Multiple Issue with Speculation



Dual Issue **without** Speculation Example

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	ld x2,0(x1)	1	2	3	4	First issue
1	addi x2,x2,1	1	5		6	Wait for ld
1	sd x2,0(x1)	2	3	7		Wait for addi
1	addi x1,x1,8	2	3		4	Execute directly
1	bne x2,x3,Loop	3	7			Wait for addi
2	ld x2,0(x1)	4	8	9	10	Wait for bne
2	addi x2,x2,1	4	11		12	Wait for ld
2	sd x2,0(x1)	5	9	13		Wait for addi
2	addi x1,x1,8	5	8		9	Wait for bne
2	bne x2,x3,Loop	6	13			Wait for addi
3	ld x2,0(x1)	7	14	15	16	Wait for bne
3	addi x2,x2,1	7	17		18	Wait for ld
3	sd x2,0(x1)	8	15	19		Wait for addi
3	addi x1,x1,8	8	14		15	Wait for bne
3	bne x2,x3,Loop	9	19			Wait for addi

Dual Issue **with** Speculation Example

Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	ld x2,0(x1)	1	2	3	4	5	First issue
1	addi x2,x2,1	1	5		6	7	Wait for ld
1	sd x2,0(x1)	2	3			7	Wait for addi
1	addi x1,x1,8	2	3		4	8	Commit in order
1	bne x2,x3,Loop	3	7			8	Wait for addi
2	ld x2,0(x1)	4	5	6	7	9	No execute delay
2	addi x2,x2,1	4	8		9	10	Wait for ld
2	sd x2,0(x1)	5	6			10	Wait for addi
2	addi x1,x1,8	5	6		7	11	Commit in order
2	bne x2,x3,Loop	6	10			11	Wait for addi
3	ld x2,0(x1)	7	8	9	10	12	Earliest possible
3	addi x2,x2,1	7	11		12	13	Wait for ld
3	sd x2,0(x1)	8	9			13	Wait for addi
3	addi x1,x1,8	8	9		10	14	Executes earlier
3	bne x2,x3,Loop	9	13			14	Wait for addi

Power Is a Limiting Factor for Multiple-Issue Processors

- Techniques to boost performance of multiple-issue processors increase power consumption more than performance.
- Issuing more instructions in parallel requires logic overhead that grows faster than the issue rate.
- There is a growing gap between peak issue rates and sustained issue rates.
- Speculation is inherently inefficient since it can never be perfect.

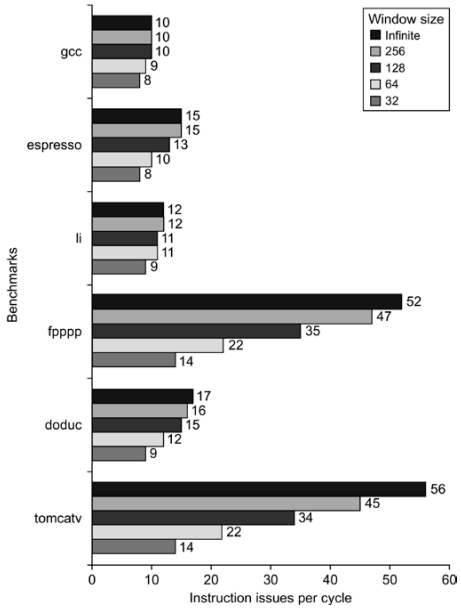
Five Primary Approaches Used in Multiple Issue Processors

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

Aggressive Processor

- 128 physical registers.
- Tournament predictor with 1K entries and a 16-entry return predictor.
- Perfect disambiguation of memory references.
- No cache misses.
- 64 instruction issues and dispatches per cycle.

ILP Available with Varying Window Size



Perfect Processor

- Infinite number of virtual registers available for register renaming. So all WAW and WAR hazards are avoided.
- Branch and jump prediction is perfect and infinite instruction prefetch buffer. So an unbounded buffer of instructions is available for execution.
- All memory addresses are known exactly. So loads can be moved up before stores whenever possible (or vice versa).
- No cache misses.
- Unlimited number of functional units.
- Only one cycle latencies for operations.

Limits Preventing a Perfect Processor

- The number of functional units must be limited.
- The number of comparisons to check for issue dependences in a window of instructions grows quadratically ($n^2 - n$). The available parallelism is reduced as the window size is decreased.
- Branch prediction levels can be high, but never perfect.
- Cannot have an infinite number of registers, but can do pretty well with a reasonable number.
- Perfect alias analysis is impossible at compile-time and very expensive at run-time.
- Cache misses and functional unit latencies must also be taken into account.
- FP operations and integer multiplies and divides cannot realistically be performed in a single cycle.

Number of Issue Comparisons for an *n* Instruction Window

- The number of comparisons to check for true dependences when issuing a window of n instructions is $n^2 - n$.

rd1 = rs1 oper rt1;

rd2 = rs2 oper rt2; rd1==rs2 || rd1==rt2 [2]

rd3 = rs3 oper rt3; rd1==rs3 || rd1==rt3 || rd2 == rs3 || rd2 == rt3 [4]

...

rdn = rsn oper rtn; [2n–2]

2n–2 + 2n–4 + ... + 4 + 2 = number of comparisons

2(n–1 + n–2 + ... + 2 + 1) =

2((n–1)*n/2) =

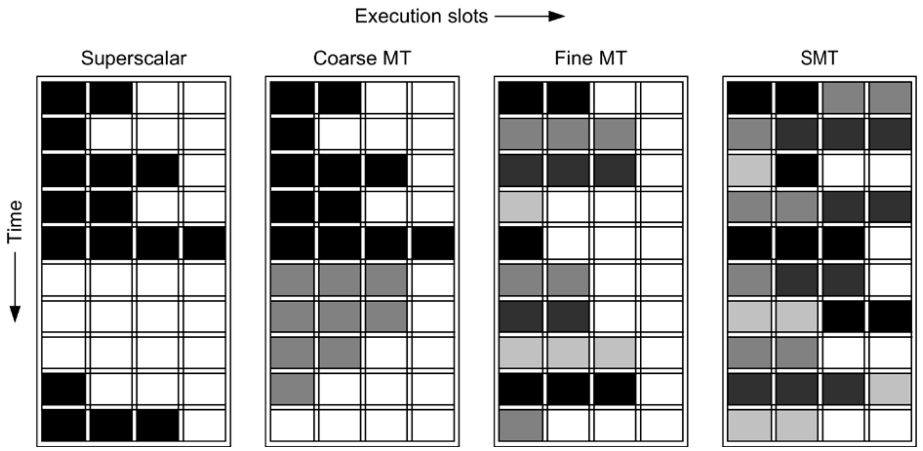
(n–1)*n =

n^2 – n

Multithreading

- Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion.
- A processor must save the state of each process (register file, PC, stack pointer, condition codes) on a context switch.
- Thread switches must be more efficient than process switches. Each thread has its own dedicated register file, PC, stack pointer, but share the same code, global variables, and heap within a process.
- types of multithreading
  - *Coarse-grained multithreading* is conventional multithreading and it switches threads on costly stalls (e.g. L2 misses).
  - *Fine-grained multithreading* switches between threads on each clock cycle in a mostly round-robin fashion.
  - *Simultaneous multithreading* (SMT) allows multiple threads to simultaneously execute by exploiting the resources of a multiple-issue, dynamically scheduled processor.

Four Approaches for Utilizing Superscalar Issue Slots



SMT Design Challenges

- Dealing with a larger register file.
- Instruction issue is more complex, which could affect the clock cycle.
- Instruction commit is more complex.
- There may be more cache and TLB conflicts, which could degrade performance.

Fallacies and Pitfalls

- Fallacy: Processors with lower CPIs will always be faster.
- Fallacy: Processors with faster clock rates will always be faster.
- Pitfall: Sometimes bigger and dumber is better.