Chapter 3: Instruction Level Parallelism (ILP) and its exploitation

- Pipeline CPI = Ideal pipeline CPI + stalls due to hazards
  - invisible to programmer (unlike process level parallelism)
- ILP: overlap execution of unrelated instructions
  - invisible to programmer (unlike process level parallelism)
- Parallelism within a basic block is limited (a branch every 3-6 instructions)
  - Hence, must explore ILP across basic blocks
- May explore loop level parallelism (fake control dependences) through
  - Loop unrolling (static, compiler based solution)
  - Using vector processors or SIMD architectures
  - Dynamic loop unrolling
- The main challenge is overcoming data dependencies

(1)

Types of dependences

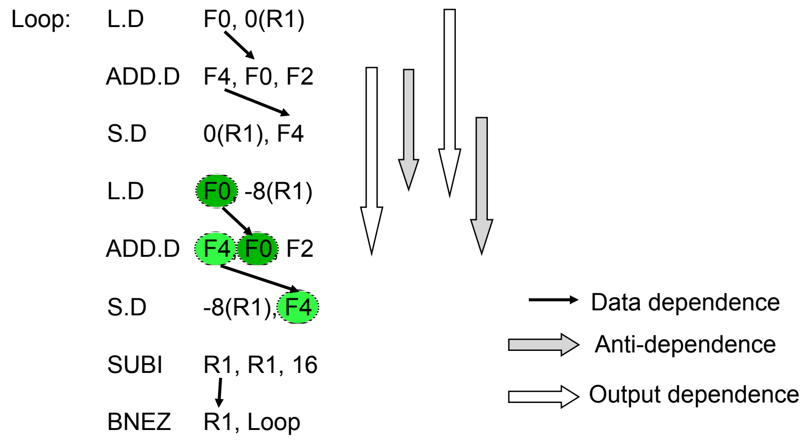
- **True data dependences:** may cause RAW hazards.
  - Instruction *Q* uses data produced by instruction *P* or by an instruction which is data dependent on *P*.
  - dependences are properties of programs, while hazards are properties of architectures.
  - easy to determine for registers but hard to determine for memory locations since addresses are computed dynamically

    EX: is 100(R1) the same location as 200(R2) or even 100(R1)?
- **Name dependences:** two instructions use the same name but do not exchange data (no data dependency**)**
  - **Anti-dependence**: Instruction *P* reads from a register (or memory) followed by instruction *Q* writing to that register (or memory).

    May cause WAR hazards.
  - **Output dependence**: Instructions *P* and *Q* write to the same location.

    May cause WAW hazards.

(2)



```
Loop:  L.D     F0, 0(R1)
       ADD.D   F4, F0, F2
       S.D     0(R1), F4
       L.D     F0 -8(R1)
       ADD.D   F4 F0 F2
       S.D     -8(R1), F4
       SUBI    R1, R1, 16
       BNEZ    R1, Loop
```

→ Data dependence

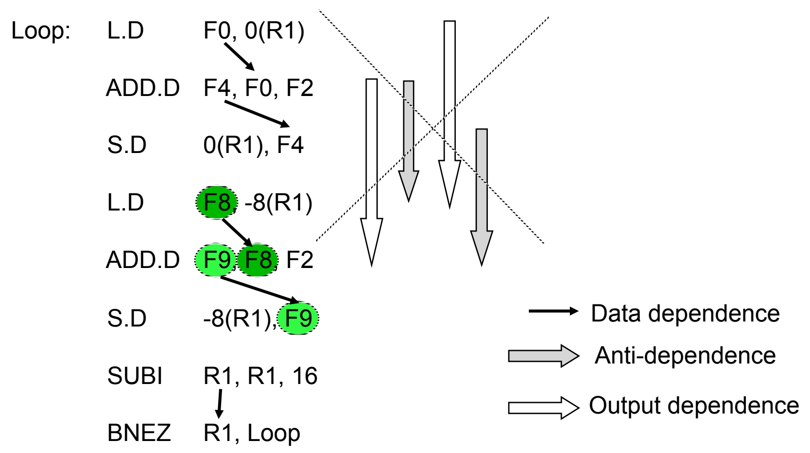
⇨ Anti-dependence

⇨ Output dependence

How can you remove name dependences?

Rename the dependent uses of F0 and F4

Register renaming



```
Loop:  L.D     F0, 0(R1)
       ADD.D   F4, F0, F2
       S.D     0(R1), F4
       L.D     F8 -8(R1)
       ADD.D   F9 F8 F2
       S.D     -8(R1), F9
       SUBI    R1, R1, 16
       BNEZ    R1, Loop
```

→ Data dependence

⇨ Anti-dependence

⇨ Output dependence

How can you remove name dependences?

Rename the dependent uses of F0 and F4

Control dependences

- Determine the order of instructions with respect to branches.

```
if P1 then S1 ;
if P2 then S2 ;
```

S1 is control dependent on P1 and
S2 is control dependent on P2 (and P1 ??).

- An instruction that is control dependent on *P* cannot be moved to a place where it is no longer control dependent on *P* , and visa-versa

Example 1:

```
    DADDU  R1,R2,R3
    BEQZ   R4,L
    DSUBU  R1,R1,R6
L:  …
    OR     R7,R1,R8
```

Example 2:

```
     DADDU  R1,R2,R3
     BEQZ   R12,skip
     DSUBU  R4,R5,R6
     DADDU  R5,R4,R9
skip:
     OR R7,R8,R9
```

OR instruction depends on the execution flow

Possible to move DSUBU before the branch (if R4 is not used after skip)

Loop carried dependences

```
For i=1,100
   a[i+1] = a[i]  + c[i]  ;
```

There is a loop carried dependence since the statement in an iteration depends on an earlier iteration.

```
For i=1,100
   a[i] = a[i]  + s ;
```

There is no loop carried dependence

- The iterations of a loop can be executed in parallel if there are no *loop carried dependences*.

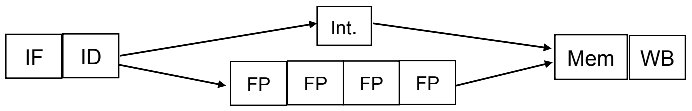
The effect of data dependencies (hazards)

- Consider the following loop, which assumes that an array starts at location 8, with a number of elements (x 8) stored in register R1.

```
Loop:   L.D      F0, 0(R1)      ; Load an element
        ADD.D    F4, F0, F2     ; add a scalar, in F2, to the element
        S.D      F4, 0(R1)      ; store result
        DADDUI   R1, R1, #-8    ; update loop index, R1
        BNE      R1, R2, Loop   ; branch if visited all elements
```

- Assume a MIPS pipeline with the following latencies
  - Latency = 3 cycles if an **FP ALU op** follows an **FP ALU op**.
  - Latency = 2 cycles if an **FP store** follows an **FP ALU op**.
  - Latency = 1 cycle if an **FP ALU op** follows an **FP load**.
  - Latency = 1 cycle if a **BNE** follows an **Integer ALU op.**

BNE cond. Evaluated in ID



(7)

Pipeline stalls due to data hazards

```
Loop:   L.D      F0, 0(R1)      ; Load an element
        stall
        ADD.D    F4, F0, F2     ; add a scalar to the array element
        stall
        stall
        S.D      F4, 0(R1)      ; store result
        DADDUI   R1, R1, #-8    ; update loop index, R1
        stall
        BNE      R1, R2, Loop   ; branch if visited all elements
```

9 clock cycles per iteration

(8)

Basic compiler techniques for exposing ILP (§3.2)

Reorder the statements :

```
Loop:   L.D     F0, 0(R1)
        DADDUI  R1, R1, #-8
        ADD.D   F4, F0, F2
        stall
        stall
        S.D     F4, 8(R1)
        BNE     R1, R2, Loop
```

7 clock cycles per iteration

(9)

Loop Unrolling (assume no pipelining)

```
Loop:   L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        DADDUI  R1, R1, #-8
        BNE     R1, R2, Loop
```

Save 0.5 instruction per iteration

```
Loop:   L.D     F0, 0(R1)
        L.D     F6, -8(R1)
        ADD.D   F4, F0, F2
        ADD.D   F8, F6, F2
        DADDUI  R1, R1, #-16
        S.D     F4, 16(R1)
        S.D     F8, 8(R1)
        BNE     R1, R2, Loop
```

Save 1 instruction per iteration

- Need to worry about boundary cases (strip mining??)
- Can reorder the statements if we use additional registers.
- What limits the number of times we unroll a loop?
- Note that loop iterations were independent

```
    for i = 1, 100
      x(i) = x(i) + c
```

(10)

Executing the unrolled loop on a pipeline

```
Loop:  L.D     F0, 0(R1)
       L.D     F6, -8(R1)
       ADD.D   F4, F0, F2
       ADD.D   F8, F6, F2
       DADDUI  R1, R1, #-16
       S.D     F4, 16(R1)
       S.D     F8, 8(R1)
       BNE     R1, R2, Loop
```

Problem if one cycle is required between integer op. and Store?.

- Can solve the problem by
  - Stalling for one cycle
  - Replacing 16(R1) in the first SD statement by 0(R1).

4 clock cycles per iteration

- What do you do if the latency = 3 cycles when S.D follows ADD.D?

(11)

Static branch prediction (§3.3)

- Static branch prediction (built into the architecture)
  - ➢The default is that branches are not taken
  - ➢The default is that branches are taken
- It is reasonable to assume that
  - ➢forward branches are often not taken
  - ➢backward branches are often taken
- May come up with more accurate predictors based on branch directions.
- Profiling is the standard technique for predicting the probability of branching.

(12)

Dynamic Branch Prediction

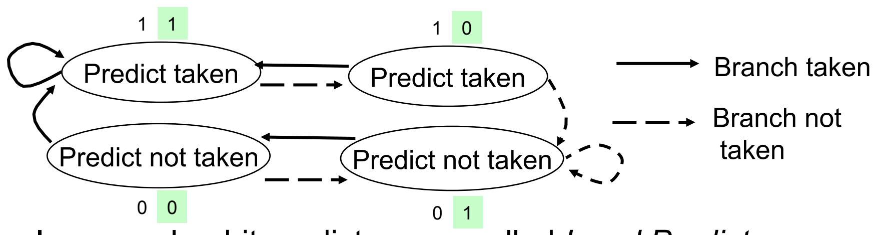
- Different than static branch predictions (predict taken or predict not taken).
- Performance depends on the accuracy of prediction and the cost of miss-prediction.
- **Branch prediction buffer (Branch history table - BHT)**:
  - 1-bit table (cache) indexed by the lower order bits of the address of the branch instructions (can be accessed in decode stage)
  - Says whether or not the branch was taken last time
  - needs to apply hashing techniques -- may have collision.
  - Will cause two miss-predictions in a loop (at start and end of loop).

```
L1:   …….
L2:   …….
      …….
      BEQZ R2, L2
      …….
      BEQZ R1, L1
```

(13)

Two bits branch predictors

- change your prediction only if you miss-predict twice
- helps if a branch changes directions occasionally (ex. Nested loops)



- In general, *n*-bit predictors are called *Local Predictors*.
  - Use a saturated counter (++ on correct prediction, -- on wrong prediction)
  - n-bit prediction is not much better than 2-bit prediction (n > 2).
  - A BHT with 4K entries is as good as an infinite size BHT
  - Dynamic branch prediction does not help in the 5-stage pipeline (why?)
  - Miss-predict when gets the wrong branch in BHT or a wrong prediction.

(14)

Correlating Branch predictors (global predictors)

- Hypothesis: recent branches are correlated (behavior of recently executed branches affects prediction of current branch).
- Example 1:

```
if (a == 2)                  SUBUI R3,R1,2
  a = 0 ;                    BNEZ  R3, L1 ...     <= B1
if (b == 2)                  ADD   R1, R0, R0
  b = 0;              L1:    SUBUI R3, R2, 2
if (a != b) ...              BNEZ  R3, L2         <= B2
                             ADD   R2, R0, R0
                      L2:    SUBU  R3, R1, R2
                             BNEZ  R3, L3         <= B3
```

If B1 and B2 are taken, then B3 will probably not be taken,
If B1 and B2 are not taken, the B3 is taken

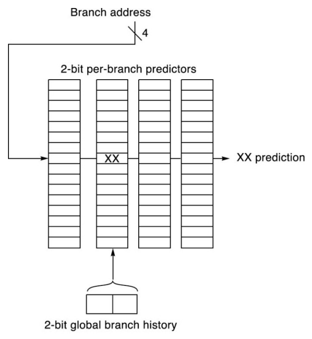
- Example 2:

```
If (d == 0) d = 1 ;
if (d == 1) .....
```

(15)

Correlating Branch predictors

- Keep history of the *m* most recently executed branches in an *m*-bit shift register.
- Record the prediction for each branch instruction, and each of the 2^m combinations.
- In general, (*m*,*n*) predictor means record last *m* branches to select between 2^m history tables each with *n*-bit predictor.
- Simple access scheme (double indexing).
- A (*0*,*n*) predictor is a local *n*-bit predictor.
- Size of table is $N\ n\ 2^m$, where *N* is the number of table entries.
- There is a tradeoff between *N* (determines collision), *n* (accuracy of local predicion) and *m* (determines history).

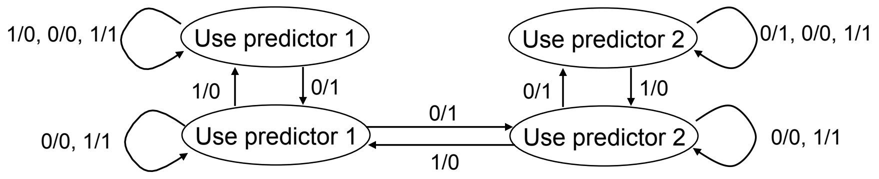


A (2,2) predictor

(16)

Tournament predictor (hybrid local-global predictors)

- Combines a global predictor and a local predictor with a strategy for selecting the appropriate predictor (multilevel predictors).

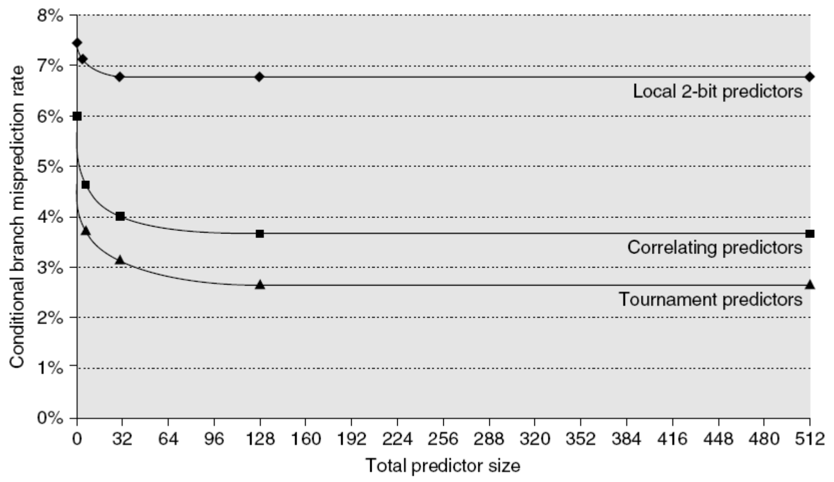


p_1/p_2 == predictor 1 is correct/predictor 2 is correct

- The Alpha 21264 selects between
  - a (12,2) global predictor with 4K entries
  - a local predictor which selects a prediction based on the outcome of the last 10 executions of any given branch.

(17)

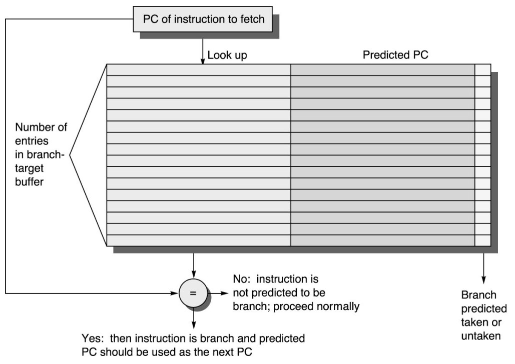
Performance of Branch predictors



(18)

Branch target buffers(§3.9)

- Store the address of the branch's target, in addition to the prediction.


PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

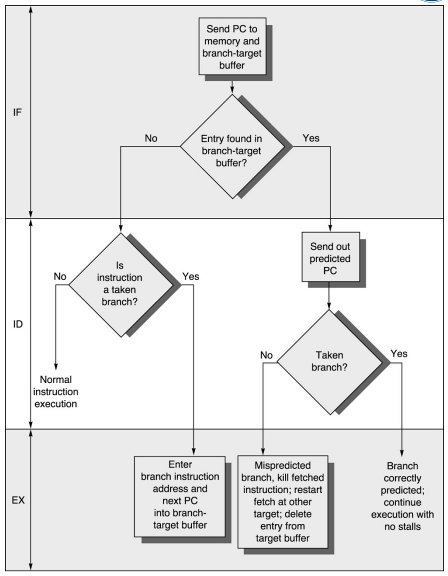
For (*m*,*n*) prediction need to keep target PC and prediction

- Can determine the target address while fetching the branch instruction
  - how do you even know that the instruction is a branch?
  - can't afford to use wrong branch address due to collision -- why?

(19)

Evaluation example:

- Assume
  - branch condition determined in ID
  - branch address determined in EX stage
  - access branch target buffer in IF stage
- what is the branch penalty if:
  - penalty for correctly predicting = 0 cycle
  - penalty for incorrectly predicting = 2 cycles
  - penalty if cannot predict and the branch is taken = 2 cycles
  - prediction accuracy = 90%,
  - branch taken frequency = 60%
  - buffer hit rate = 90%
- may store the target instruction and not only the address - useful when access of table needs more than one cycle.


IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

© 2003 Elsevier Science (USA). All rights reserved.

Dynamically scheduled pipelines (§3.4 - 3.5)

Key idea: allow instructions behind stall to proceed

DIV.D	F0, F2, F4	Stall
ADD.D	F10, F0, F8	
SUB.D	F12, F8, F14	

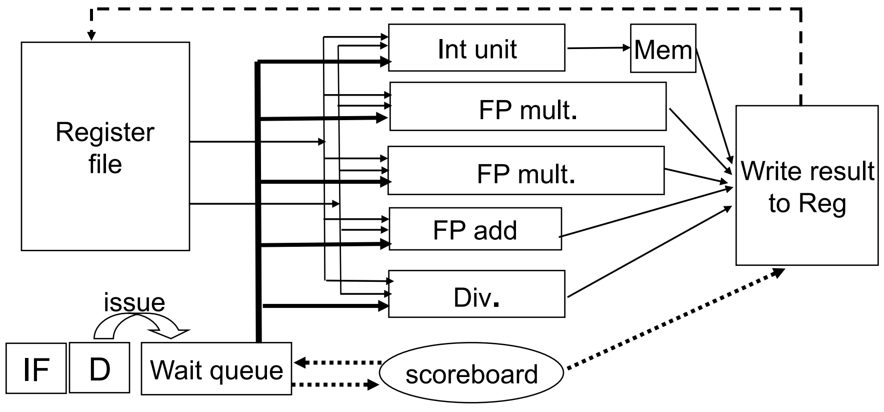
- Enables out-of-order execution,
- Can lead to out-of-order completion.

Using Scoreboards (see Appendix C.7):

- Dates to the first supercomputer, the CDC 6600 in 1963
- Split the ID stage into
    - Issue - decode and check for structural hazards,
    - Read operands - wait until no data hazards, then read operands.
- Instructions wait in a queue and may move to the EX stage out of order.

(21)

A scoreboard architecture



- The scoreboard is responsible for instruction issue and execution, including hazard detection. It is also controlling the writing of the results.
- The "Wait queue" does not actually exist -- it is implemented as a table inside the scoreboard.

(22)

Scoreboard information

- **Instruction status**:
  - issued, read operands and started execution, completed execution or wrote result,
- **Functional unit status** (assuming non-pipelined units)
  - busy/not busy
  - operation (if unit can perform more than one operation)
  - destination register - F_i
  - source registers (containing source operands) - F_j and F_k
  - the unit producing the source operands (if stall to avoid RAW hazards) - Q_j and Q_k
  - flags to indicate that source operands are ready -- R_j and R_k
- **Register result status**:
  - indicates the functional unit that contains an instruction which will write into each register (if any)

(23)

Four stages of scoreboard control

- **Issue only if no structural, WAR or WAW hazards.**
  - Issue if functional unit is free and
    - » the execution units do not contain an instruction which writes to the destination register (to avoid WAW)
    - » No issued instruction (in the wait queue) will read from the destination register (to avoid WAR)
  - otherwise, stall, and block subsequent instructions
  - the fetch unit stalls when the queue between the fetch and the issue stages is full (may be only one buffer).
- **Read operands only if no RAW hazard.**
  - If a RAW hazard is detected, wait until the operands are ready,
  - When the operands are ready, read the registers and move to the execution stage,
  - note that instructions may proceed to the EX stage out-of-order.
- **Execution.**
  - When execution terminates, notify the score board.
- **Write result to register file**

(24)

Example:

Instruction status

Instruction	Issue	Read op.	Exec. Completed	Write result
L.D F6, 34(R2)	X	X	X	X
L.D F2, 45(R3)	X	X	X	
MUL.D F0, F2, F4	X			
SUB.D F8, F6, F2	X			
DIV.D F10, F0, F6	X			
ADD.D F6, F8, F2				

Func. unit status

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

Register status

	F0	F2	F4	F6	F8	F10	F12	…	F30
Func. U	Mult1	Int.			Add	Div			

(25)

Example:

Instruction status

Instruction	Issue	Read op.	Exec. Completed	Write result
L.D F6, 34(R2)	X	X	X	X
L.D F2, 45(R3)	X	X	X	X
MUL.D F0, F2, F4	X			
SUB.D F8, F6, F2	X			
DIV.D F10, F0, F6	X			
ADD.D F4, F8, F2				

Func. unit status

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

Register status

	F0	F2	F4	F6	F8	F10	F12	…	F30
Func. U	Mult1				Add	Div			

(26)

Example: when MUL.D is ready to write

Instruction status

Instruction		Issue	Read op.	Exec. Completed	Write result
L.D	F6, 34(R2)	X	X	X	X
L.D	F2, 45(R3)	X	X	X	X
MUL.D	F0, F2, F4	X	X	X	
SUB.D	F8, F6, F2	X	X	X	X
DIV.D	F10, F0, F6	X			
ADD.D	F4, F8, F2	X	X	X	

Func. unit status

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

Register status

	F0	F2	F4	F6	F8	F10	F12	…	F30
FU	Mult1		Add			Div			

(27)

Limitations of the scoreboard approach

- No forwarding
- do not issue on structural hazards
- Wait until WAW and WAR hazards are cleared
- Did not discuss control hazards

Limitations of the scoreboard in general

- The number of parallel buses between registers and pipeline units (determine the number of issues per cycles).
- The number of scoreboard entries.

Need to extend the scoreboard to the case where the execution units are pipelined?

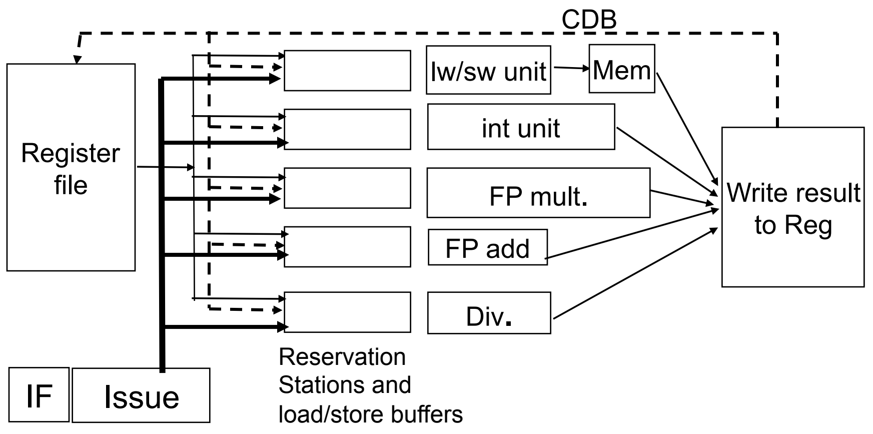
(28)

The Tomasulo approach

- Introduced for IBM 360/91 in 1966
- Main improvements over the scoreboard approach:
  - Uses forwarding on a Common Data Bus (CDB) -- more efficient dealing with RAW hazards,
  - avoids WAR hazards by reading the operands in the instruction-issue order, instead of stalling at issue. To accomplish this an instruction reads an available operand before waiting for the other.
  - Avoids WAW hazards by renaming the registers (using the *id* of a reservation station rather than the register *id*)
  - The control information and logic are distributed to the functional unit and not centralized.

(29)

The architecture for the Tomasulo scheme



- Each reservation station has an *id* and is used by one instruction during the lifetime of this instruction.
- Each unit has one or more reservation stations
- Reservation stations play the role of temporary registers (renaming)

(30)

Book keeping in Tomasulo's algorithm

- **Instruction status**:
  - issue, execute or write result,
- **Reservation stations (functional units) status**:
  - busy/not busy
  - operation (if unit can perform more than one operation)
  - source operands (data values) - V_j and V_k
  - the *reservation stations* producing the source operands (if stall to avoid RAW hazards) - Q_j and Q_k
  - Address field, *A*, for load/store buffers (store effective address)
- **Register result status**:
  - indicates the *reservation station* that contains an instruction which will write into each register (if any)

(31)

Three stages of control

- **Issue**
  - If a reservation station is available for the needed functional unit
    - » read ready operands
    - » for operands that are not ready, rename the register to the reservation station that will produce it,
  - Store/load operands are issued if a buffer (reservation station) is available.
- **Execution.**
  - Monitor the CDB for the operand that is not ready,
  - When both operands are available, execute.
  - If more than one station per unit, only one unit can start execution.
  - Do not start execution before previous branches have completed.
- **Write result.**
  - Write to CDB (and to registers) -- may be a structural hazards if only one CDB bus.
  - Make the reservation station (the functional unit) available.

(32)

Load and store instructions:

- Uses load/store buffers, and each buffer is like a reservation station.
- Address calculation (put result in buffer) + memory operation
- The result of a load is put on the CDB
- Stores are executed in program order (loads in any order)
- Performs memory disambiguation between store and load buffers,

Remarks:

- May have more reservation stations than registers (a large virtual register space)
- The original Tomasulo algorithm was introduced before caches were incorporated into commercial processors
- If more than one issued instruction writes into a register, only the last one does the actual write (no WAW hazards).

(33)

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

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	Y	Load					45+Reg[R3]
Add1	Y	Sub		Mem[34+Reg[R2]]	Load2		
Add2	Y	Add			Add1	Load2	
Add3	no						
Mult1	Y	Mul		Reg[F4]	Load2		
Mult2	Y	Div		Mem[34+Reg[R2]]	Mult1		

	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Mult1	load2		Add2	Add1	Mult2			

(34)

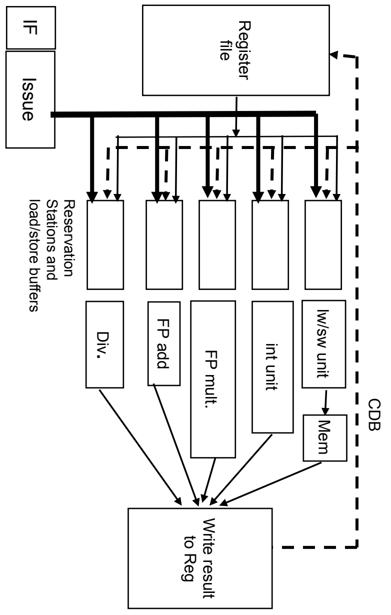
Instruction		Issue	Execute	Write result
L.D	F6, 34(R2)	X	X	X
L.D	F2, 45(R3)	X	X	X
MUL.D	F0, F2, F4	X	X	
SUB.D	F8, F2, F6	X	X	X
DIV.D	F10, F0, F6	X		
ADD.D	F6, F8, F2	X	X	X

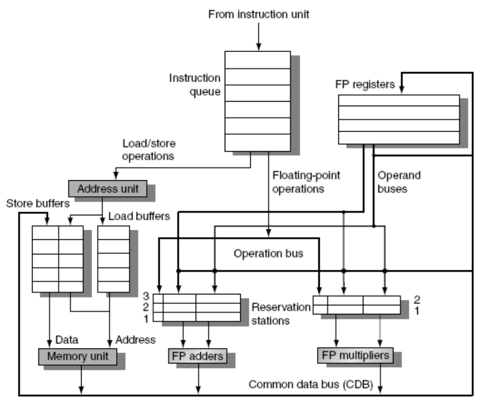
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	no						
Add1	no						
Add2	no						
Add3	no						
Mult1	Y	Mul	Mem[45+Reg[R3]]	Reg[F4]			
Mult2	Y	Div		Mem[34+Reg[R2]]	Mult1		

	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Mult1					Mult2			

(35)

Figure 3.6





(36)

Instruction	Issue	Execute	Write result
L.D F0, 0(R1)	X	X	
MUL.D F4, F0, F2	X		
S.D F4, 0(R1)	X		
L.D F0, 8(R1)	X	X	
MUL.D F4, F0, F2	X		
S.D F4, 8(R1)	X		

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	y	ld					Reg[R1]+0
Load2	y	ld					Reg[R1]+8
store1	y	sd	Reg[R1]+0			Mult1	
store2	y	sd	Reg[R1]+8			Mult2	
Add	no						
Mult1	Y	Mul		Reg[F2]	Load1		
Mult2	Y	Mul		Reg[F2]	Load2		

	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Load2		store 2						

(37)

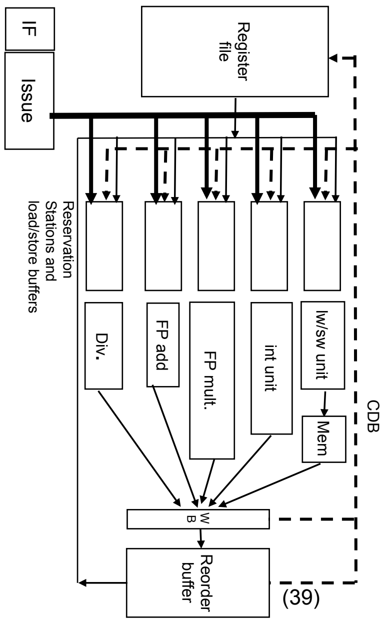
Hardware-based Speculation (§3.6)

- The goal is to allow instructions after a branch to start execution before the outcome of the branch is confirmed.
- There should be *no* consequences (including exceptions) if it is determined that the instruction should not execute.
  - Use dynamic branch prediction and use OOO execution
  - Use a Reorder Buffer (ROB) to reorder instructions after execution
  - Commit results to registers and memory in-order
  - All un-committed results can be flushed if a branch is miss-predicted
  - Service interrupts only when an instruction is ready to commit
- Should free the reservation station when an instruction is in the reorder buffer.
- For each register, R, the status table keeps the ROB number reserved by the instruction which will write into R (instead of the RES station number).

(38)

Reorder buffers (ROB)

- 3 fields: instruction, destination, value
- When issuing a new instruction, read a register value from the ROB if the status table indicates that the source instruction is in a ROB.
- Hence, ROBs supply operands between execution complete & commit => more virtual registers.
- ROBs form a circular queue ordered in the “issue order”.
- Once an instruction reaches the head of the ROB queue, it commits the results into register or memory.
- Hence, it’s easy to undo speculated instructions on miss-predicted branches or on exceptions
- Should flush the pipe as soon as you discover a miss-predictions – all earlier instructions should commit (problems??)



(39)

Steps of Speculative Tomasulo Algorithm

Combining branch prediction with dynamic scheduling

- **Issue** (sometimes called Dispatch)
  - If a RES station and a ROB are free, issue the instruction to the RES station after reading ready registers and renaming non-ready registers
- **Execution** (sometimes called issue)
  - When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute (checks RAW)
- **Write result** (WB)
  - Write on CDB to all awaiting RES stations & send the instruction to the ROB; mark reservation station available.
- **Commit** (sometimes called graduation)
  - When instruction is at head of ROB, update registers (or memory) with result and free ROB. A miss-predicted branch flushes all non-committed instructions.

(40)

Example:

- Assume that MUL.D just entered the ROB
- MUL.D is at the head of the ROB ready to commit
- SUB.D and ADD.D have been in the ROB before MUL.D
- DIV.D is in RES station waiting for the result of MUL.D

Instruction	Issued	Execute	In ROB	committed
L.D F6, 34(R2)	X	X	X	x
L.D F2, 45(R3)	X	X	X	x
MUL.D F0, F2, F4	X	X	X	
SUB.D F8, F2, F6	X	X	X	
DIV.D F10, F0, F6	X	X		
ADD.D F6, F8, F2	X	X	X	

(41)

Reservation stations status

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest.	A
Load1	no							
Load2	no							
Add1	Y	sub	Mem[45+Reg[R3]]	Mem[34+Reg[R2]]			ROB4	
Add2	Y	add		Mem[45+Reg[R3]]	ROB4		ROB6	
Add3	no							
Mult1	Y	Mul	Mem[45+Reg[R3]]	Reg[F4]			ROB3	
Mult2	Y	Div		Mem[34+Reg[R2]]	ROB3		ROB5	

ROB status

Name	Busy	Instruction	State	Dest.	value
ROB1	no	L.D F6, 34(R2)	Commit	F6	xxx
ROB2	no	L.D F2, 45(R3)	Commit	F2	xxx
ROB3	yes	MUL.D F0, F2, F4	Write result	F0	xxx
ROB4	yes	SUB.D F8, F2, F6	Write result	F8	xxx
ROB5	yes	DIV.D F10, F0, F6	Issued/executing	F10	
ROB6	yes	ADD.D F6, F8, F2	Write result	F6	xxx

Register status

	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	ROB3			ROB6	ROB4	ROB5			

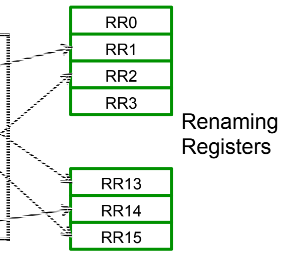
(42)

Renaming Registers

- Common variation of speculative design
- Reorder buffer keeps instruction information but not the result
- Extend register file with extra *renaming registers* to hold speculative results
- Rename register allocated at issue; result is written into renaming register when execution completes; renaming register is copied to real register on commit.
- Operands read either from register file (real or speculative) or via Common Data Bus
- Advantage: operands are always from single source (extended register file)

Name	Busy	Instruction	State	Dest.	R Reg.
ROB1	no	L.D F6, 34(R2)	Commit	F6	1
ROB2	no	L.D F2, 45(R3)	Commit	F2	13
ROB3	yes	MUL.D F0, F2, F4	Write result	F0	15
ROB4	yes	SUB.D F8, F2, F6	Write result	F8	2
ROB5	yes	DIV.D F10, F0, F6	Issued/executing	F10	
ROB6	yes	ADD.D F6, F8, F2	Write result	F6	14

ROB status



(43)

Multiple issue processors (§3.7)

- Issue more than one instruction per clock cycle
  - VLIW processors (static scheduling by the compiler)
  - Superscalar processors
    - » Statically scheduled (in-order execution)
    - » Dynamically scheduled (out-of-order execution)
- results in CPI < 1 (Instructions per clock, IPC > 1)
- The fetch unit gets an *issue packet* which contains multiple instructions
- The issue unit issues 1-8 instructions every cycle (usually, the issue unit is itself pipelined)
  - independent instructions
  - multiple resources should be available
  - branch prediction should be accurate
- Leads to multiple instruction completion per cycle

(44)

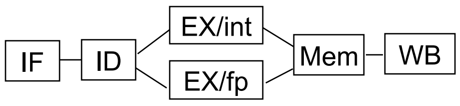
Five primary approaches

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 ARM Coretex A8
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

(45)

A statically scheduled MIPS superscalar

- Assume two execution pipes
  - one for load, store, branch and integer operations
  - one for floating point operations
  - IF fetches 2 instructions
  - ID process 2 instructions
  - increase load on register file



Type	*Pipe*	*Stages*					
Int. instruction	IF	ID	EX	MEM	WB		
FP instruction	IF	ID	EX	MEM	WB		
Int. instruction		IF	ID	EX	MEM	WB	
FP instruction		IF	ID	EX	MEM	WB	
Int. instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

(46)

Loop Unrolling in Superscalar

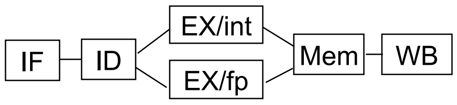
Assume: LD to ADDD: 1 Cycle and ADDD to SD: 2 Cycles

	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	L.D F0,0(R1)		1
	L.D F6,-8(R1)		2
	L.D F10,-16(R1)	ADD.D F4,F0,F2	3
	L.D F14,-24(R1)	ADD.D F8,F6,F2	4
	L.D F18,-32(R1)	ADD.D F12,F10,F2	5
	S.D F4, 0(R1)	ADD.D F16,F14,F2	6
	S.D F8, -8(R1)	ADD.D F20,F18,F2	7
	S.D F12, -16(R1)		8
	S.D F16, -24(R1)		9
	DADDI R1,R1,-40		10
	BNEZ R1,Loop		11
	S.D F20, -32(R1)		12

- **Unrolled 5 times to avoid delays.**
- **12 clocks, or 2.4 clocks per iteration.**

(47)

MIPS superscalar

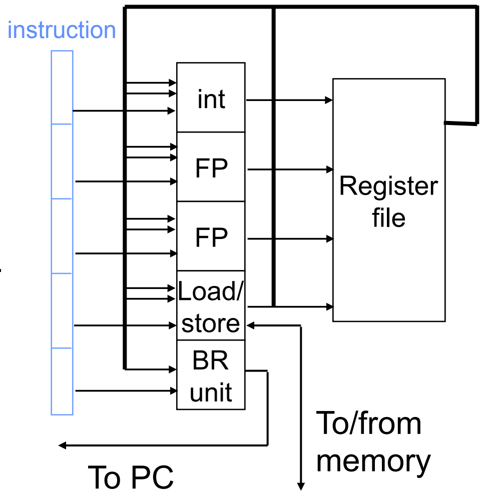


- While Integer/FP split is simple for the HW, we can get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards
- If two instructions issued every cycle, greater difficulty of decode and issue
  - Even 2 units => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue
- 1 cycle load delay expands to 3 instructions in SS
  - instruction in right half can't use it, nor instructions in next slot
- Control hazards is twice as expensive

(48)

VLIW architectures

- Compiler packs multiple, independent, operations into one single instruction.
- E.g.,
  - 1 integer operations,
  - 2 FP ops,
  - 1 Memory refs,
  - 1 branch.
  - instruction = 5*24 = 120 bits.
- Lock-step issue of instructions (one per cycle)
- In a pure VLIW approach, the compiler resolves all hazards (resolved by hardware in MIPS superscalars)



(49)

Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				2
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		3
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
		ADD.D F20,F18,F2	ADD.D F24,F22,F2		5
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2			6
S.D F12,-16(R1)	S.D F16,-24(R1)				7
S.D F20,-32(R1)	S.D F24,-40(R1)			DADDI R1,R1,-48	8
S.D F28,-0(R1)				BNEZ R1,Loop	9

- Unrolled 7 times
- 7 results in 9 clocks, or 1.3 clocks per iteration.
- Average: 2.5 ops per clock, 50% efficiency
- Note: Need more registers in VLIW (15 vs. 11 in SS)

(50)

Dynamic scheduling (§3.8)

- Extends Tomasulo's algorithm to issue two (or more) instructions simultaneously to reservation stations.
- Either issue an instruction every half clock cycle, or double the logic to handle two instructions at once.
- Use the same logic. Some restrictions may be used to simplify hardware.
  - Example: issue only one FP and one int. operation every clock cycle. This reduces the load on the register files.
  - Do not issue dependent instructions in the same cycle
- To deal with control hazards without speculation (no ROBs): instructions following a branch can be issued but cannot start execution before the branch is resolved.
- Will look at the execution of

```
L1:  L.D       F0, 0(R1)
     ADD.D     F4, F0, F2
     S.D       F4, 0(R1)
     DADDIU    R1, R1, -8
     BNE       R1, R2, L1
```

(51)

Dual issue with one int. and one FP add unit (not pipelined). Latency = 1, 2 and 3 for int. operations, loads and FP adds.

Iteration	Instruction	Issued at	Executes	Mem access	Write CDB	comments
1	L.D F0, 0(R1)	1	2	3	4	First issue
1	ADD.D F4, F0, F2	1	5		8	Wait for L.D
1	S.D F4, 0(R1)	2	3	9		Wait for ADD.D
1	DADDIU R1, R1, -8	2	4		5	Wait for ALU
1	BNE R1, R2, L1	3	6			wait for DADDIU
2	L.D F0, 0(R1)	4	7	8	9	Wait for BNE
2	ADD.D F4, F0, F2	4	10		13	Wait for L.D
2	S.D F4, 0(R1)	5	8	14		Wait for ADD.D
2	DADDIU R1, R1, -8	5	9		10	Wait for ALU
2	BNE R1, R2, L1	6	11			wait for DADDIU
3	L.D F0, 0(R1)	7	12	13	14	Wait for BNE
3	ADD.D F4, F0, F2	7	15		18	Wait for L.D
3	S.D F4, 0(R1)	8	13	19		Wait for ADD.D
3	DADDIU R1, R1, -8	8	14		15	Wait for ALU
3	BNE R1, R2, L1	9	16			wait for DADDIU

- No speculation: L.D cannot start before BNE completes execution (cycle 6)
- Instruction after BNE cannot be issued in same cycle: branch not yet predicted
- L.D, S.D, DADDIU and BNE use the same integer unit.

(52)

If we have a separate int. unit for memory address calculation and we have a second CDB.

Iteration	Instruction	Issued at	Executes	Mem access	Write CDB	comments
1	L.D F0, 0(R1)	1	2	3	4	First issue
1	ADD.D F4, F0, F2	1	5		8	Wait for L.D
1	S.D F4, 0(R1)	2	3	9		Wait for ADD.D
1	DADDIU R1, R1, -8	2	**3**		**4**	Exec. earlier
1	BNE R1, R2, L1	3	5			wait for DADDIU
2	L.D F0, 0(R1)	4	6	7	8	Wait for BNE
2	ADD.D F4, F0, F2	4	9		12	Wait for L.D
2	S.D F4, 0(R1)	5	7	13		Wait for ADD.D
2	DADDIU R1, R1, -8	5	**6**		7	Exec. earlier
2	BNE R1, R2, L1	6	8			wait for DADDIU
3	L.D F0, 0(R1)	7	9	10	11	Wait for BNE
3	ADD.D F4, F0, F2	7	12		15	Wait for L.D
3	S.D F4, 0(R1)	8	10	16		Wait for ADD.D
3	DADDIU R1, R1, -8	8	9		10	Exec. earlier
3	BNE R1, R2, L1	9	11			wait for DADDIU

No speculation: no instruction after BNE can start before BNE completes execution (cycle 5)

(53)

Multiple issue with speculation

Consider same example (as last slide)

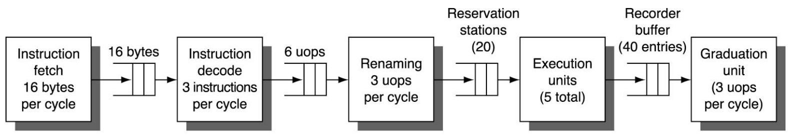
Iteration	Instruction	Issued at	Executes	Mem access	Write CDB	comments
1	L.D F0, 0(R1)	1	2	3	4	First issue
1	ADD.D F4, F0, F2	1	5		8	Wait for L.D
1	S.D F4, 0(R1)	2	3	9		Wait for ADD.D
1	DADDIU R1, R1, -8	2	3		4	
1	BNE R1, R2, L1	3	5			wait for DADDIU
2	L.D F0, 0(R1)	~~4~~ 3	~~6~~ 4	~~7~~ 5	~~8~~ 6	No wait for BNE
2	ADD.D F4, F0, F2	4	~~9~~ 7		~~12~~ 10	Wait for L.D
2	S.D F4, 0(R1)	~~5~~ 4	~~7~~ 5	~~13~~ 10		Wait for ADD.D
2	DADDIU R1, R1, -8	5	6		7	
2	BNE R1, R2, L1	~~6~~ 5	~~8~~ 6			wait for DADDIU
3	L.D F0, 0(R1)	~~7~~ 6	~~9~~ 7	~~10~~ 8	~~11~~ 9	No wait for BNE
3	ADD.D F4, F0, F2	~~7~~ 6	~~12~~ 10		~~15~~ 13	Wait for L.D
3	S.D F4, 0(R1)	~~8~~ 7	~~10~~ 8	~~16~~ 14		Wait for ADD.D
3	DADDIU R1, R1, -8	~~8~~ 7	~~9~~ 8		~~10~~ 9	
3	BNE R1, R2, L1	~~9~~ 8	~~11~~ 9			wait for DADDIU

Allow speculation, and may issue a branch with another instruction

(54)

Dynamic Scheduling in the P6 microarchitecture

- Doesn’t pipeline the IA-32 instructions (complex instructions)
- Decode unit translates the Intel instructions into MIPS-like micro-operations, called *μops* (most instructions translate to 1 – 4 *μ* ops).
- Complex IA-32 instructions are executed by a conventional microprogram that issues long sequences of *μ* ops
- Sends *μ ops* to reorder buffer & reservation stations


Instruction fetch 16 bytes per cycle
16 bytes
Instruction decode 3 instructions per cycle
6 uops
Renaming 3 uops per cycle
Reservation stations (20)
Execution units (5 total)
Recorder buffer (40 entries)
Graduation unit (3 uops per cycle)

- A 14-stage pipeline:
  - 8 stages for in-order fetch of IA-32 instructions, generating *μ* ops, decoding, dynamic branch prediction and dispatch of *μ* ops
  - 3 stages for execution into 5 pipelined units (from 1 to 32 pipelined stages)
  - 3 stages for instruction commit.

(55)

Limits to Multi-Issue Machines

- 1 branch in 5: How to keep a 5-issue processor busy?
- Latencies of units: many operations must be scheduled
- Need about (Pipeline Depth x No. Functional Units) of independent instructions.
- Need More instruction fetch bandwidth (easy)
- Need Duplicate FUs to get parallel execution (easy)
- Need more ports in Register File (hard) and more memory bandwidth (harder)
- Impact of decoding multiple instructions on clock rate and pipeline depth.
- Decode issue in Superscalar: how wide is practical?
- More logic lead to larger power consumption ➔ lower performance per watt.

(56)

Return address prediction

- 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

Integrated instruction Fetch unit that performs

- Branch prediction
- Instruction prefetch

Energy Efficiency

- Speculation is only energy efficient when it significantly improves performance

Value Prediction

- Loads that load from a constant pool
- Instruction that produces a value from a small set of values
- Not incorporated into modern processors

(57)

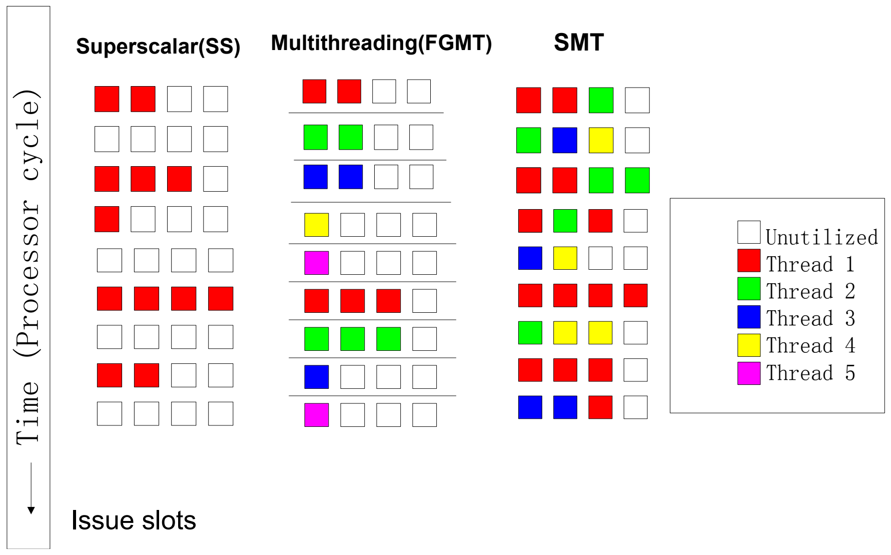
Simultaneous Multithreading (§3.10)

- Key idea

  Issue multiple instructions from multiple threads each cycle

- Features
  - Fully exploit thread-level parallelism and instruction-level parallelism.
  - Better Performance for
    - Mix of independent programs
    - Programs that are parallelizable

(58)



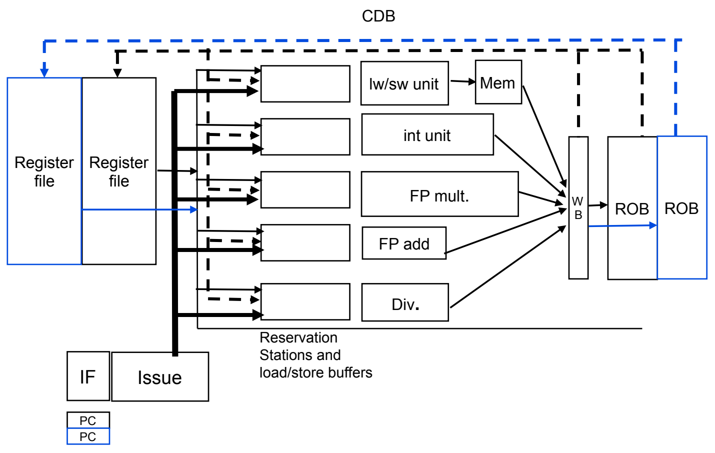
(59)

SMT ARCHITECTURE

- Base Processor: like out-of-order superscalar processor. [MIPS R10000]
- With N simultaneous running threads, need N PC and more than N*32 physical registers for register renaming in total.
- Need large register files, longer register access time → pipeline stages are added
- Share the cache hierarchy and branch prediction hardware.
- Cache and branch prediction interference, and increased memory pressure.

(60)

SMT Architecture



Reservation Stations and load/store buffers

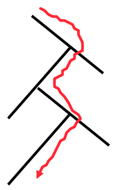
(61)

Left over slides

(62)

Trace Scheduling

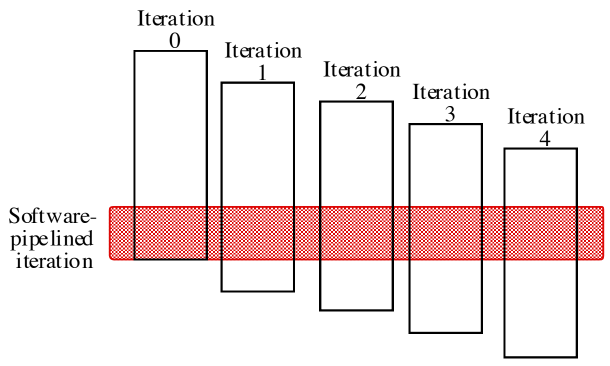
- Parallelism across IF branches vs. LOOP branches
- Two steps:
  - – *Trace Selection*
    - » Find likely sequence of basic blocks (*trace*) of (statically predicted or profile predicted) long sequence of straight-line code
  - – *Trace Compaction*
    - » Squeeze trace into few VLIW instructions
    - » Need bookkeeping code in case prediction is wrong
- In essence, we are predicting the trace path
- Compiler undoes bad guess (discards values in registers)
- Subtle compiler bugs mean wrong answer vs. poor performance;



(63)

Software Pipelining

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (Tomasulo in SW)



(64)

Software Pipelining Example

Unrolled loop

```
L:  L.D    F0,0(R1)
    ADD.D  F4,F0,F2
    S.D    F4,0(R1)
    L.D    F0,-8(R1)
    ADD.D  F4,F0,F2
    S.D    F4,-8(R1)
    L.D    F0,-16(R1)
    ADD.D  F4,F0,F2
    S.D    F4,-16(R1)
    DADDI  R1,R1,-24
    BNEQ   R1,R2,L
```

```
    L.D    F0,0(R1)
    ADD.D  F4,F0,F2
    L.D    F0,-8(R1)
    DADDI  R2,R2,16
L:  S.D    F4,0(R1)
    ADD.D  F4,F0,F2
    L.D    F0,-16(R1)
    DADDI  R1,R1,-8
    BNEQ   R1,R2,L
    S.D    F4,-8(R1)
    ADD.D  F4,F0,F2
    S.D    F4,-16(R1)
```

- Symbolic Loop Unrolling
  - Maximize result-use distance
  - Less code space than unrolling
  - Fill & drain pipe only once per loop

(65)

HW support for exposing more ILP to compilers

- **Conditionally executed (predicated) instructions**:
  if (x) then A = B op C else NOP
  - If false, then neither store result nor cause exception
  - Alpha, MIPS, PowerPC and SPARC have conditional move.
  - Drawbacks: Still takes a clock even if "annulled" and Stall if condition is evaluated late
- **Boosting**: hardware is available to "undo" a wrong speculation.
  - Instructions may be marked "speculative" and boosted above branches
  - When instruction no longer speculative, write boosted results (*instruction commit*) or discard boosted results
  - execute out-of-order but commit in-order to prevent irrevocable action (update state or exception) until instruction commits

(66)

Intel/HP "Explicitly Parallel Instruction Computer (EPIC)"

- 3 Instructions in 128 bit "groups"; field determines if instructions dependent or independent
  - Smaller code size than old VLIW, larger than x86/RISC
  - Groups can be linked to show independence > 3 instr
- 64 integer registers + 64 floating point registers
  - Not separate files per functional unit as in old VLIW
- Hardware checks dependencies
  (interlocks => binary compatibility over time)
- Predicated execution (select 1 out of 64 1-bit flags)
  => 40% fewer mispredictions?
- IA-64 : name of instruction set architecture; EPIC is a machine type

(67)