Montek Singh

Mon, Oct 3, 2005

Topic: ***Instruction-Level Parallelism***

(Dynamic Scheduling: Introduction)

Instruction-Level Parallelism

Relevant Book Reading (HP3):

- Dynamic Scheduling (in hardware): Appendix A & Chapter 3
- Compiler Scheduling (in software): Chapter 4

Hardware Schemes for ILP

- Why do it in hardware at run time?
  - Works when can't know dependences at compile time
  - Simpler compiler
  - Code for one machine runs well on another machine
- Key idea: Allow instructions behind stall to proceed

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

  - Enables out-of-order execution
  - Implies out-of-order completion
  - ID stage check for both structural and data dependences

Dynamic Scheduling

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

- 7-cycle divider
- 4-cycle adder

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
In-order	**DIV.D F0, F2, F4**	F	D	**I**	**E**	E	E	E	E	E	E	M	W								
	ADD.D F10, F0, F8		F	D	**I**	I	I	I	I	I	I	**E**	E	E	E	M	W				
	SUB.D F12, F8, F14			F	D	**I**	I	I	I	I	I	I	I	I	I	**E**	E	E	E	M	W
Out-of-order	**DIV.D F0, F2, F4**	F	D	**I**	**E**	E	E	E	E	E	E	M	W								
	ADD.D F10, F0, F8		F	D	**I**	I	I	I	I	I	I	**E**	E	E	E	M	W				
	SUB.D F12, F8, F14			F	D	**I**	**E**	E	E	E	M	W									

- Instructions are issued in order (leftmost *I*)
- Execution can begin out of order (leftmost *E*)
- Execution can terminate out of order (*W*)
- What is *I*?

Explanation of *I*

- To be able to execute the SUB.D instruction
  - A function unit must be available
    - Adder is free in example
  - There should be no data hazards preventing early execution
    - None in this example
  - We must be able to recognize the two previous conditions
    - Must examine several instructions before deciding on what to execute
- I represents the ***instruction window*** (or ***issue window***) in which this examination happens
  - If every instruction starts execution in order, then ***I*** is superfluous
  - Otherwise:
    - Instruction enter the issue window in order
    - Several instructions may be in issue window at any instant
    - Execution can begin out of order

Out-of-order Execution and Renaming

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

- WAW hazard on register F10: prevents out-of-order execution on machine like CDC 6600
- If processor was capable of ***register renaming:***
  - the WAW hazard would be eliminated
    - SUB.D could execute early as before
  - example: IBM 360/91

Memory Consistency

- Memory consistency refers to the order of main memory accesses as compared to the order seen in sequential (unpipelined) execution
  - Strong memory consistency: All memory accesses are made in strict program order
  - Weak memory consistency: Memory accesses may be made out of order, provided that no dependences are violated
- Weak memory consistency is more desirable
  - leads to increased performance
- In what follows, ignore register hazards

- Q: When can two memory accesses be re-ordered?

Load-Load

```
LW  R1, (R2)
LW  R3, (R4)
```

Load-Store

```
LW  R1, (R2)
SW  (R3), R4
```

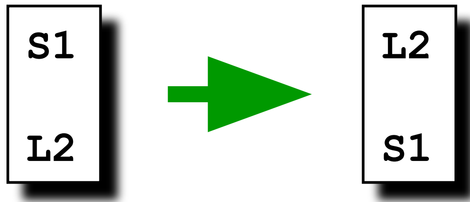
Store-Store

```
SW  R1, (R2)
SW  R3, (R4)
```

Store-Load

```
SW  (R1), R2
LW  R3, (R4)
```

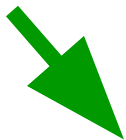
- Load-Load can always be interchanged (if no `volatile`s)
- Load-Store and Store-Store are never interchanged
- Store-Load is the only promising program transformation
  - Load is done earlier than planned, which can only help
  - Store is done later than planned, which should cause no harm
- Two variants of transformation
  - If load is independent of store, we have load bypassing
  - If load is dependent on store through memory (e.g., (R1) == (R4)), we have load forwarding



Load Bypassing

```
S1: M[R1] ← R2

L2: R4 ← M[R1]
```



```
R4 ← R2

M[R1] ← R2
```

Load Forwarding

- Either transformation can be performed at compile time if the memory addresses are known, or at run-time if the necessary hardware capabilities are available
- Compiler performs load bypassing in loop unrolling example (next lecture)
- In general, if compiler is not sure, it should not do the transformation
- Hardware is never “not sure”

Load Bypassing in Hardware

- Requires two separate queues for LOADs and STOREs
- Every LOAD has to be checked for every STORE waiting in the store queue to determine whether there is a hazard on a memory location
  - assume that processor knows original program order of all these memory instructions
- In general, LOAD has priority over STORE
- For the selected LOAD instruction, if there exists a STORE instruction in the store queue such that …
  - LOAD is behind STORE (in program order), and
  - their memory addresses are the same

  … then the LOAD cannot be sent to memory, and must wait to be executed only after the store is executed

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
(0) R0 = M(u)	I	E	E	E	E																				
(1) M(v) = R1		I	-	-	-	-	-	-	-	-	-	-	-	E	E	E	E								
(2) R2 = R3+R4			I	E	E	E																			
(3) R5 = M(w)				I	-	E	E	E	E																
(4) R6 = M(x)					I	-	-	-	-	E	E	E	E												
(5) R7 = M(v)						I	-	-	-	-	-	-	-	-	-	-	-	E	E	E	E				
(6) R8 = R9 +R10							I	E	E	E															
(7) M(w) = R11								I	-	-	-	-	-	-	-	-	-	-	-	-	-	E	E	E	E

- Memory access takes four cycles
- Actions at various points in time
  - End of cycle 1: LQ = [(0)]; SQ = [ ]; execute first load
  - End of cycle 5: LQ = [(3), (4)]; SQ = [(1)]; execute first load
  - End of cycle 9: LQ = [(4), (5)]; SQ = [(1), (7)]; execute first load
  - End of cycle 13: LQ = [(5)]; SQ = [(1), (7)]; load yields to store
- We are assuming that no LOADs or STOREs issue between instructions 7 and 22

History of Dynamic Scheduling

- First implemented on CDC 6600 and IBM 360/91 in the early 1960s
  - Fetched and decoded single instr. at a time in program order
  - Between decoding and execution, instructions stayed in issue window where hazards were detected and resolved
  - Some hazards resolved before issuing (e.g., availability of FU)
  - Most data hazards resolved only after issuing
  - Hazard resolution done with sophisticated hardware scheme
    - For each instruction in issue window, separately track, monitor, enforce, and eventually resolve all hazards affecting the instruction before it could begin execution
  - Result: Instructions started execution when they were ready to execute, rather than starting in program order