EE282
Computer Architecture

Lectures 8 & 9: Instruction-Level Parallelism

October 23-25, 2001

Andrew Wolfe
Computer Systems Laboratory
Stanford University
awolfe@csl.stanford.edu

Announcements

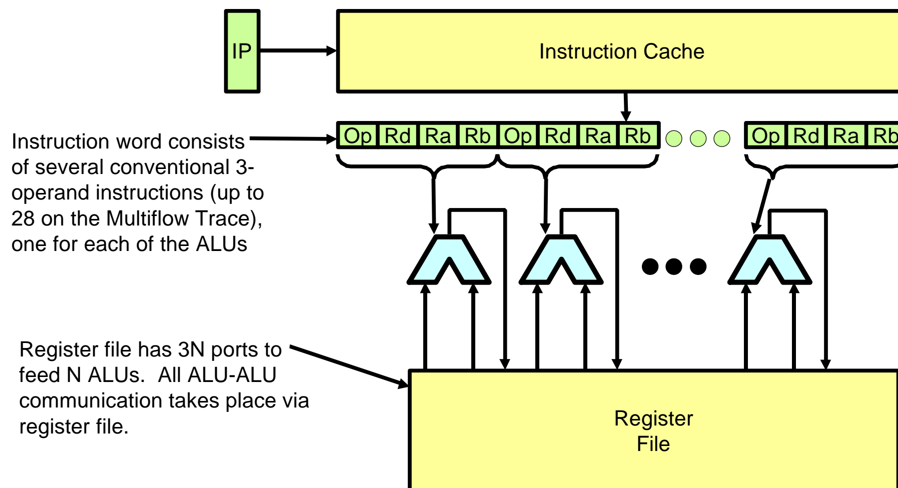
- Midterm
  - Thursday, Nov 1, 7:00pm to 9:15pm
  - Terman Aud.
  - Special Arrangements - contact me (Probably Afternoon - 10/29)
- Class Thursday - not live (tape)
  - Taping today @ 2:45 upstairs

Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

This Week's Lectures

- VLIW
- Static Scheduling Multiple Issue
- Dynamic Instruction Scheduling

Very-Long Instruction Word (VLIW) Computers



Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

A Two-Dimensional Schedule

```
for(i=0;i<n;i++) {
  d[i] = a[i]*b[i] + c ;
}
```

Two iterations unrolled

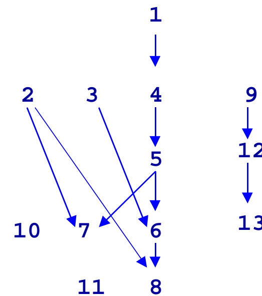
Cycle	MEM1	MEM2	MEM3	MEM4	ADD1	ADD2	ADD3	ADD4	MULT1	MULT2	MULT3	MULT4	BR
1	R1=a[i]	R2=b[i]	R3=a[i+1]	R4=b[i+1]	R10=R10+1	R11=R11+16	R12=R12+16						
2									R5=R1*R2	R6=R3*R4			
3					R7=R5+R9	R8=R6+R9	R14=R10<R15						
4	d[i]=R7	d[i+1]=R8				R13=R13+16							BNEZ 1

16 operations in 4 cycles
Average ILP = 4
Mostly NOPs, occupancy 16/52 = 31%

With max unrolling, limited by ADD to 1.75 cycles (70%)

Scheduling Code for VLIW

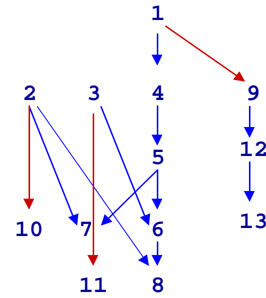
```
LOOP:
1.    LW    R4,0(R1)
2.    LD    F2,0(R2)
3.    LD    F4,0(R3)
4.    ADD   R4,R4,R3
5.    LD    F6,0(R4)
6.    MULD  F8,F6,F4
7.    MULD  F10,F6,F2
8.    ADDD  F8,F8,F2
9.    SUBI  R1,R1,#4
10.   SUBI  R2,R2,#4
11.   SUBI  R3,R3,#4
12.   SUB   R5,R1,R6
13.   BNEZ  R5,LOOP
```



Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

Scheduling Code for VLIW

```
LOOP:
1.    LW    R4,0(R1)
2.    LD    F2,0(R2)
3.    LD    F4,0(R3)
4.    ADD   R4,R4,R3
5.    LD    F6,0(R4)
6.    MULD  F8,F6,F4
7.    MULD  F10,F6,F2
8.    ADDD  F8,F8,F2
9.    SUBI  R1,R1,#4
10.   SUBI  R2,R2,#4
11.   SUBI  R3,R3,#4
12.   SUB   R5,R1,R6
13.   BNEZ  R5,LOOP
```



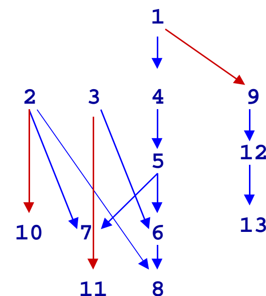
Add Antidependencies

AW/WJD EE282 Lecture 8-9 10/23-25/01 7

Scheduling Code for VLIW

3-issue slot VLIW

1	2	3
4	9	10
5	11	12
6	7	
8	13	



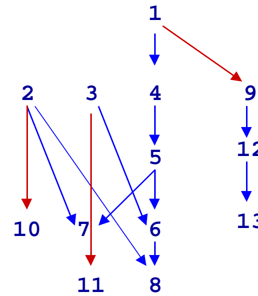
AW/WJD EE282 Lecture 8-9 10/23-25/01 8

Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

Typed issue slots

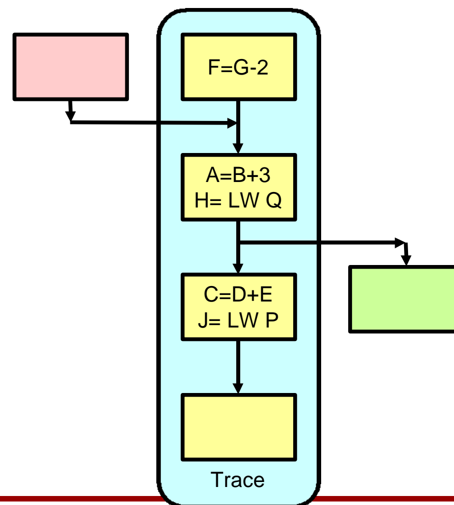
4-issue slot VLIW

IALU	FALU	LD/ST	BR
		1	
4		2	
9		5	
10	7	3	
12	6		
11	8		13



Trace Scheduling

- Easier to find ILP in a longer sequence
  - more operations to choose from
  - more parallel expressions
- Most programs have basic blocks
- Fuse several sequences together along a *trace*
- Allow code motion past basic block boundaries
  - Fixup code when entering or exiting trace
  - speculative loads above branch

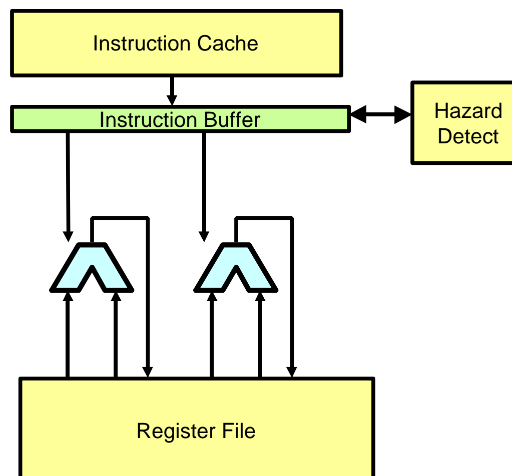


Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

VLIW Pros and Cons

- Pros
  - Very simple hardware
    - no dependency detection
    - simple issue logic
    - just ALUs and register file
  - Potentially exploits large amounts of ILP
- Cons
  - Lockstep execution (static schedule)
    - very sensitive to long latency operations (cache misses)
  - Global register file hard to build
  - Lots of NOPs
    - poor code 'density'
    - I-cache capacity and bandwidth compromised
  - Must recompile sources
  - Implementation visible through ISA

Multiple Issue



Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

Multiple Issue (Details)

- Superficially looks like VLIW but:
  - Dependencies and structural hazards checked at run-time
  - Can run existing binaries
    - must recompile for performance, not correctness
- More complex issue logic
  - Swizzle next N instructions into position
  - Check dependencies and resource needs
  - Issue M <= N instructions that can execute in parallel

AW/WJD EE282 Lecture 8-9 10/23-25/01 13

Example Multiple Issue

Issue rules: at most 1 LD/ST, at most 1 floating op

Latency: LD - 3, int-1, F*-6, F+-3

```
LOOP:  LD     F0, R1 ;         // a[i]
       LD     F2, R2 ;         // b[i]
       LD     F4, 8(R1) ;      // a[i+1]
       LD     F6, 8(R2) ;      // b[i+1]
       ADDI   R1, #16 ;
       ADDI   R2, #16 ;
       MULTD  F8, F0, F2 ;     // a[i] * b[i]
       MULTD  F10, F4, F6 ;    // a[i+1] * b[i+1]
       ADDD   F12, F8, F16 ;   // + c
       ADDD   F14, F10, F16 ;     // + c
       SD     F12, R3 ;        // d[i]
       SD     F14, 8(R3) ;     // d[i]
       ADDI   R3, #16 ;
       ADDI   R4, #2 ;         // increment i
       SLT    R5, R4, R6 ;     // i<n-1
       BNEZ   R5, LOOP ;
```

AW/WJD EE282 Lecture 8-9 10/23-25/01 14

Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

Rescheduled for Multiple Issue

Issue rules: at most 1 LD/ST, at most 1 floating op

Latency: LD - 3, int-1, F*-6, F+-3

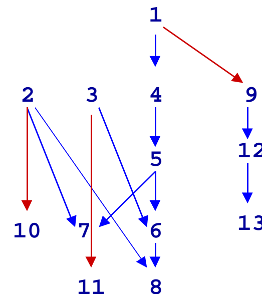
```
LOOP:  LD     F0, R1 ;        // a[i]
       ADDI   R1, #16 ;
       LD     F2, R2 ;        // b[i]
       ADDI   R2, #16 ;
       LD     F4, -8(R1) ;    // a[i+1]
       ADDI   R4, #2 ;        // increment i
       LD     F6, -8(R2) ;    // b[i+1]
       MULTD  F8, F0, F2 ;    // a[i] * b[i]
       ADDI   R3, #16 ;
       MULTD  F10, F4, F6 ;   // a[i+1] * b[i+1]
       SLT    R5, R4, R6 ;    // i<n-1
       ADDD   F12, F8, F16 ; // + c
       SD     F12, -16(R3) ; // d[i]
       ADDD   F14, F10, F16 ;    // + c
       SD     F14, -8(R3) ;  // d[i]
       BNEZ   R5, LOOP ;
```


Multiple Issue - in-order execution

4- typed issue slots

4 instruction window

IALU	FALU	LD/ST	BR
		1	
		2	
4		3	
		5	
	6		
	7		
9	8		
10			
11			
12			13



Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

Multiple Issue vs VLIW

- More complex issue logic
  - check dependencies
  - check structural hazards
  - issue variable number of instructions (0-N)
  - shift unissued instructions over
- Able to run existing binaries
  - recompile for performance, not correctness
- Datapaths identical
  - but bypass requires detection
- Neither VLIW or multiple-issue can schedule around run-time variation in instruction latency
  - cache misses
- Dealing with run-time variation requires run-time or *dynamic* scheduling

AW/WJD EE282 Lecture 8-9 10/23-25/01 17

The Problem with Static Scheduling

- In-order execution
  - an unexpected long latency blocks ready instructions from executing
  - binaries need to be rescheduled for each new implementation
  - small number of *named* registers becomes a bottleneck

```
LW   R1, C         //miss 50cy
LW   R2, D
MUL  R3, R1, R2
SW   R3, C
LW   R4, B         //ready
ADD  R5, R4, R9
SW   R5, A
LW   R6, F
LW   R7, G
ADD  R8, R6, R7
SW   R7, E
```

AW/WJD EE282 Lecture 8-9 10/23-25/01 18

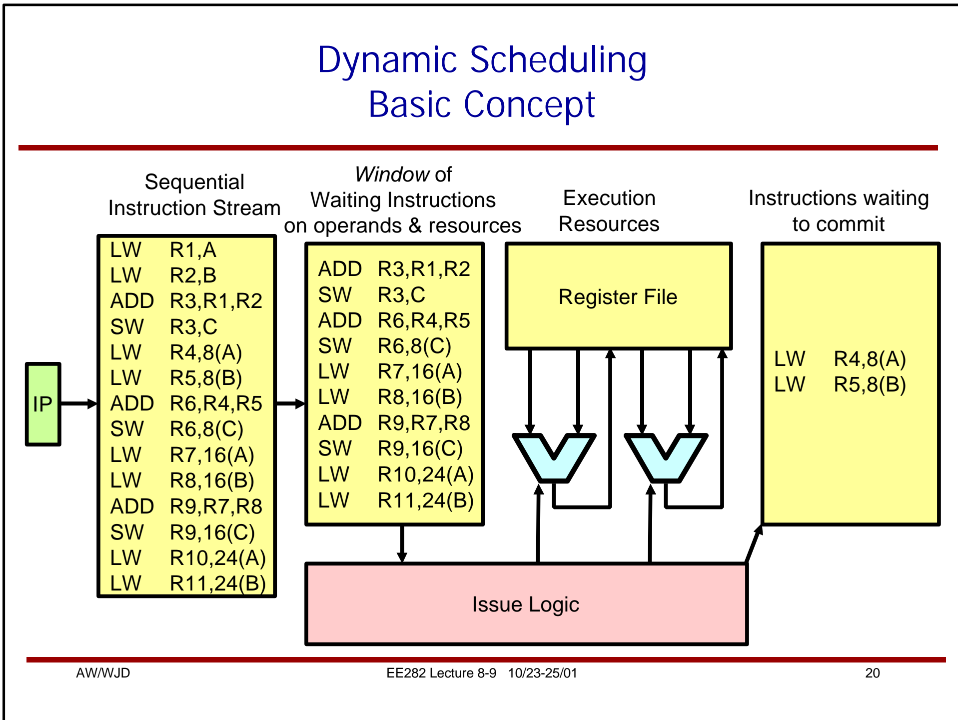
Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

Dynamic Scheduling

- Determine execution order of instructions at *run time*
- Schedule with knowledge of run-time variable latency
  - cache misses
- Compatibility advantages
  - avoid need to recompile old binaries
  - avoid bottleneck of small *named* register sets
    - but still need to deal with spills
- Significant hardware complexity

AW/WJD EE282 Lecture 8-9 10/23-25/01 19

Dynamic Scheduling Basic Concept



AW/WJD EE282 Lecture 8-9 10/23-25/01 20

Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

Dynamic Scheduling
Basic Concept (2)

- Instructions pass through 4 transitions
  - fetch: instruction is fetched from memory and placed in the *window* of *pending* instructions.
  - issue: an instruction in the window is *issued* to an execution unit when
    - a unit is available
    - all of its inputs are available
    - output register is available
  - complete: when the instruction completes execution it enters the reorder buffer to await its turn to *commit* or retire
  - commit (retire): when all previous instructions have committed without raising exceptions this instruction commits

inactive → [fetch] → pending → [issue] → executing → [complete] → waiting → [retire] → inactive

Example

Cycle	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	26	27	28	29	30
LD R1,A	I	M	X	X	X	X	X	X	X	X	X	C																		
LD R2,B		I	C																											
MUL R3,R1,R2												I	X	X	C															
LD R4,C													I	M	X	X	X	X	X	X	X	X	X	C						
LD R5,D														I	C															
MUL R6,R5,R4																								I	X	X	C			
ADD R7,R3,R6																											I	X	C	
SD R7,E																													I	C
LD R1,A	I	M	X	X	X	X	X	X	X	X	X	C	R																	
LD R2,B		I	C											R																
MUL R3,R1,R2												I	X	X	C	R														
LD R4,C			I	M	X	X	X	X	X	X	X	X	X	C			R													
LD R5,D				I	C													R												
MUL R6,R5,R4														I	X	X	C		R											
ADD R7,R3,R6																	I	X	C	R										
SD R7,E																			I	C	R									

Top shows in-order execution
Bottom shows dynamic execution
I=Issue, M=miss, X=execute, C=complete, R=retire

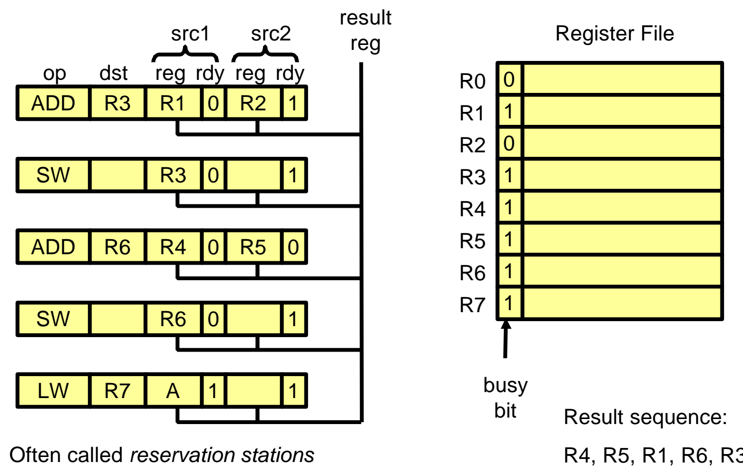
Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

Implementation Issues

- Instruction window
  - fixed number of instruction *slots* (e.g., 32)
    - generic or
    - partitioned over fn units
  - fetch next sequential instruction whenever a slot is free
  - mark input and output registers *busy*
  - slots monitor register status and execution unit reservation tables
- Issue when
  - all input operands available
  - output operand (register) not *busy* (WAW, WAR) due to earlier instruction
  - execution unit is available
- Commit when
  - all previous instructions have committed

AW/WJD EE282 Lecture 8-9 10/23-25/01 23

Implementing A Simple Instruction Window



Often called *reservation stations*

Result sequence:

R4, R5, R1, R6, R3

AW/WJD EE282 Lecture 8-9 10/23-25/01 24

Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

Implementing a Simple Instruction Window (2)

- Add an instruction to the window
  - only when dest register is not busy
  - mark destination register busy
  - check status of source registers and set ready bits
- When each result is generated
  - compare dest register field to all waiting instruction source register fields
  - update ready bits
  - mark dest register not busy
- Issue an instruction when
  - execution resource is available
  - all source operands are ready
- Result
  - issues instructions out of order as soon as source registers are available
  - allows only one operation in the window per destination register

Register Renaming (1)

What about this sequence?

1	LW	R1, 0(R4)
2	ADD	R2, R1, R3
3	LW	R1, 4(R4)
4	ADD	R5, R1, R3

Can't add 3 to the window since R1 is already busy

Need 2 R1s!

Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

Register Renaming (2)

S1	LW	R1	data	1		1
S2	ADD	R2	S1	0	data	1
S3	LW	R1	data	1		1
S4	ADD	R5	S2	0	data	1

	busy bit	tag	
R1	1	S3	
R2	1	S2	
R3	0		
R3	0		
R5	1	S4	

Add a *tag* field to each register

Add instruction to window even if dest register is busy

When adding instruction to window
- read data of non-busy source registers and retain
- read tags of busy source registers and retain
- write tag of destination register with slot number

When result generated
- compare *tag* of result to not-ready source fields
- grab data if match

Example Execution

```
LW   R1, 0(R2)
ADD  R1, R1, R1
SW   R1, 0(R2)
ADD  R1, R3, R3
SW   R1, 4(R2)
ADD  R1, R2, R2
SW   R1, 8(R2)
```

Slot	Op	Dst	Src1	R	Src2	R
S1						
S2						
S3						
S4						
S5						
S6						
S7						

	B	Tag	Value
R1			
R2			4
R3			5
R4			
R5			
R6			
R7			

Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

Example Execution
All Instructions Fetched, None Executed

```
LW   R1, 0(R2)
ADD  R1, R1, R1
SW   R1, 0(R2)
ADD  R1, R3, R3
SW   R1, 4(R2)
ADD  R1, R2, R2
SW   R1, 8(R2)
```

Slot	Op	Dst	Src1	R	Src2	R
S1	LW	R1	4	1		
S2	ADD	R1	S1	0	S1	0
S3	SW		S2	0	4	1
S4	ADD	R1	5	1	5	1
S5	SW		S4	0	4	1
S6	ADD	R1	4	1	4	1
S7	SW		S6	0	4	1

	B	Tag	Value
R1	1	S6	
R2			4
R3			5
R4			
R5			
R6			
R7			

Example Execution
Execute S1, S4, and S6

```
LW   R1, 0(R2)
ADD  R1, R1, R1
SW   R1, 0(R2)
ADD  R1, R3, R3
SW   R1, 4(R2)
ADD  R1, R2, R2
SW   R1, 8(R2)
```

Slot	Op	Dst	Src1	R	Src2	R
S1	LW	R1	4	1		
S2	ADD	R1	3	1	3	1
S3	SW		S2	0	4	1
S4	ADD	R1	5	1	5	1
S5	SW		10	1	4	1
S6	ADD	R1	4	1	4	1
S7	SW		8	1	4	1

	B	Tag	Value
R1		S6	8
R2			4
R3			5
R4			
R5			
R6			
R7			

Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

Example Execution
Retire S1, Execute S2, S5, S7

```
LW   R1, 0(R2)
ADD  R1, R1, R1
SW   R1, 0(R2)
ADD  R1, R3, R3
SW   R1, 4(R2)
ADD  R1, R2, R2
SW   R1, 8(R2)
```

Slot	Op	Dst	Src1	R	Src2	R
S1	LW	R1	4	1		
S2	ADD	R1	3	1	3	1
S3	SW		6	1	4	1
S4	ADD	R1	5	1	5	1
S5	SW		10	1	4	1
S6	ADD	R1	4	1	4	1
S7	SW		8	1	4	1

	B	Tag	Value
R1		S6	8
R2			4
R3			5
R4			
R5			
R6			
R7			

Addr	Data
4	3
8	10
0xC	8

Example Execution
Retire S2, Execute S3

```
LW   R1, 0(R2)
ADD  R1, R1, R1
SW   R1, 0(R2)
ADD  R1, R3, R3
SW   R1, 4(R2)
ADD  R1, R2, R2
SW   R1, 8(R2)
```

Slot	Op	Dst	Src1	R	Src2	R
S1	LW	R1	4	1		
S2	ADD	R1	3	1	3	1
S3	SW		6	1	4	1
S4	ADD	R1	5	1	5	1
S5	SW		10	1	4	1
S6	ADD	R1	4	1	4	1
S7	SW		8	1	4	1

	B	Tag	Value
R1		S6	8
R2			4
R3			5
R4			
R5			
R6			
R7			

Addr	Data
4	6
8	10
0xC	8

Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

Some Issues

- How do we *rename* several (2-4) instructions per cycle?
- How do we make sure that the correct value winds up in the register?
- How do we make sure events (exceptions) are handled in the right order?
- When can we move a load past a store?

Retirement and Re-order Buffers

- Must *commit* instructions in order
  - check exceptions
  - update visible register state
  - update memory
- Maintain *slots* as a circular buffer
  - commit instruction at head when it is finished
  - fetch new instructions to tail

	Slot	Op	Dst	Src	R	Src	R
	S1						
Head →	S2	ADD	R1	S1	0	S1	0
	S3	SW		S2	0	R2	1
	S4	ADD	R1	R3	1	R3	1
	S5	SW		S4	0	R2	1
	S6	ADD	R1	R2	1	R2	1
	S7	SW		S6	0	R2	1
Tail →	S8						

Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.

Dynamic Scheduling and Memory Operations

- Store cannot update memory until instruction commits
  - but value can be used by subsequent loads before commit
- A load cannot execute before a preceding store unless they are known to be to different addresses
  - disambiguation
  - hard at compile time, easy at run time

L/S	Addr	Data

Memory Conflict Resolution
(Memory Order Buffer)

Some History, Dynamic Scheduling Then and Now

- IBM 360/91
  - Reservation stations (register renaming)
  - Tomasulo's algorithm
  - optimized for storage-to-register instructions
- CDC 6600
  - Scoreboard
- Intel P6 (Pentium Pro/Pentium II)
  - Converts CISC instructions to one or more RISC instructions
  - Reservation stations (register renaming)
  - In-order retirement

Copyright (c) 1999-2001 by W. J. Dally, A. Wolfe, all rights reserved.