Lecture 12: Dynamic Branch Prediction, Superscalar, VLIW, and Software Pipelining

Professor Randy H. Katz

Computer Science 252

Spring 1996

Review: Tomasulo Summary

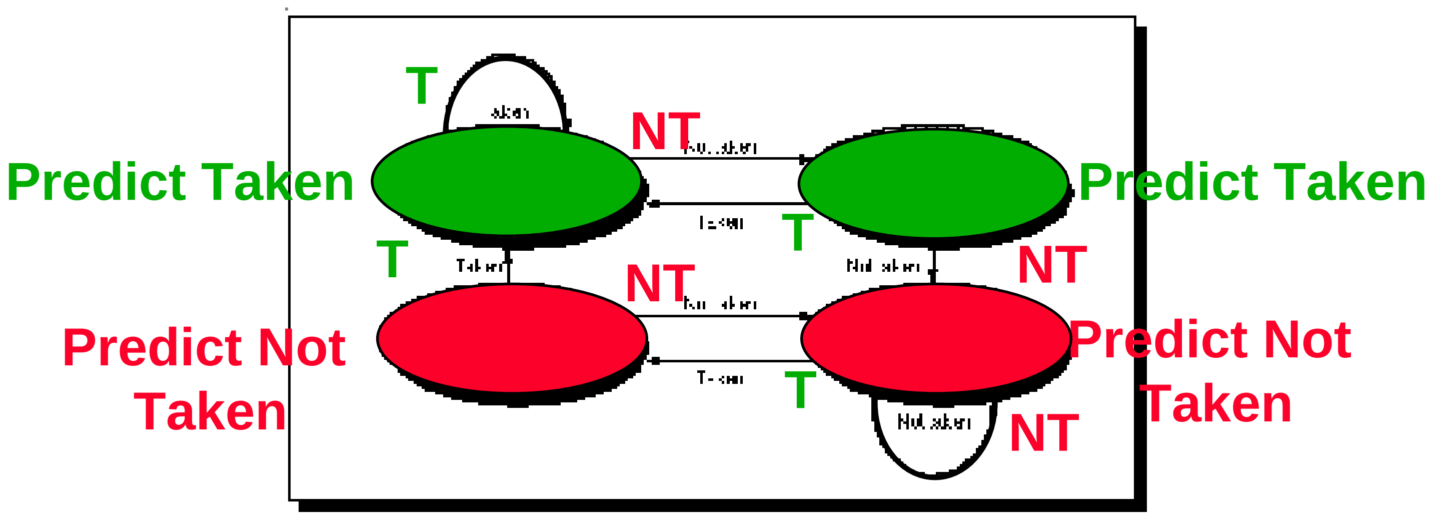
- Registers not the bottleneck
- Avoids the WAR, WAW hazards of Scoreboard
- Not limited to basic blocks (provided branch prediction)
- Allows loop unrolling in HW
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- Next: More branch prediction

Dynamic Branch Prediction

- Performance = *f*(accuracy, cost of misprediction)
- Branch History Table is simplest
  - Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
- Problem: in a loop, 1-bit BHT will cause two mispredictions:
  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts exit instead of looping

Dynamic Branch Prediction

- **Solution: 2-bit scheme where change prediction only if get misprediction *twice:* (Figure 4.13, p. 264)**



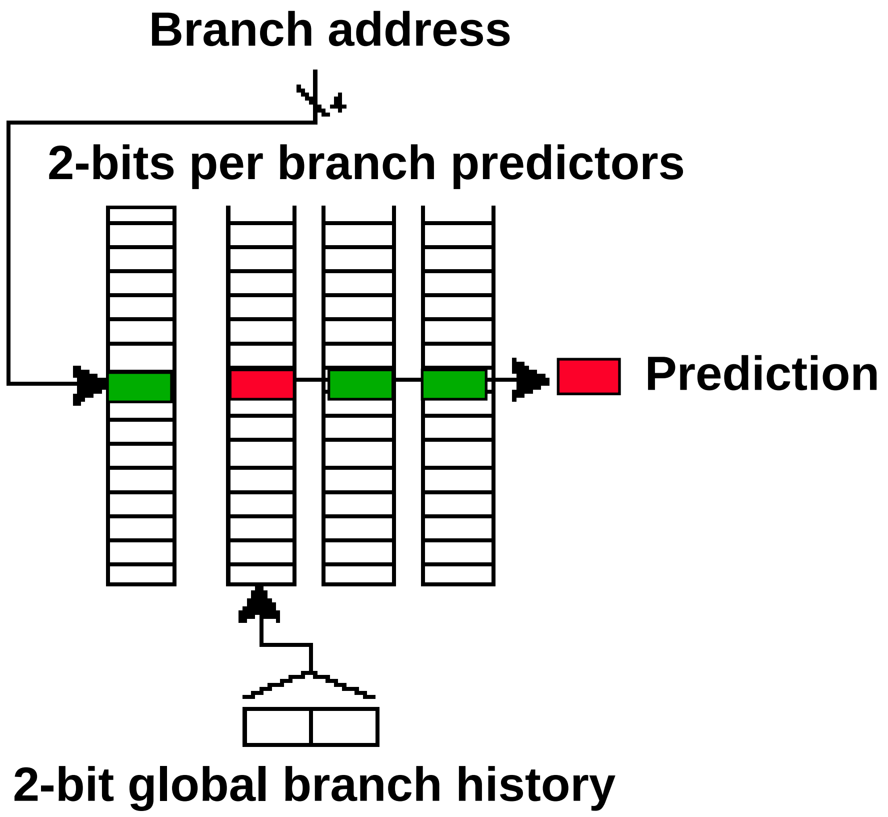
BHT Accuracy

- Mispredict because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table
- 4096 entry table programs vary from 1% misprediction (nasa7, tomcatv) to 18% (eqntott), with spice at 9% and gcc at 12%
- 4096 about as good as infinite table, but 4096 is a lot of HW

Correlating Branches

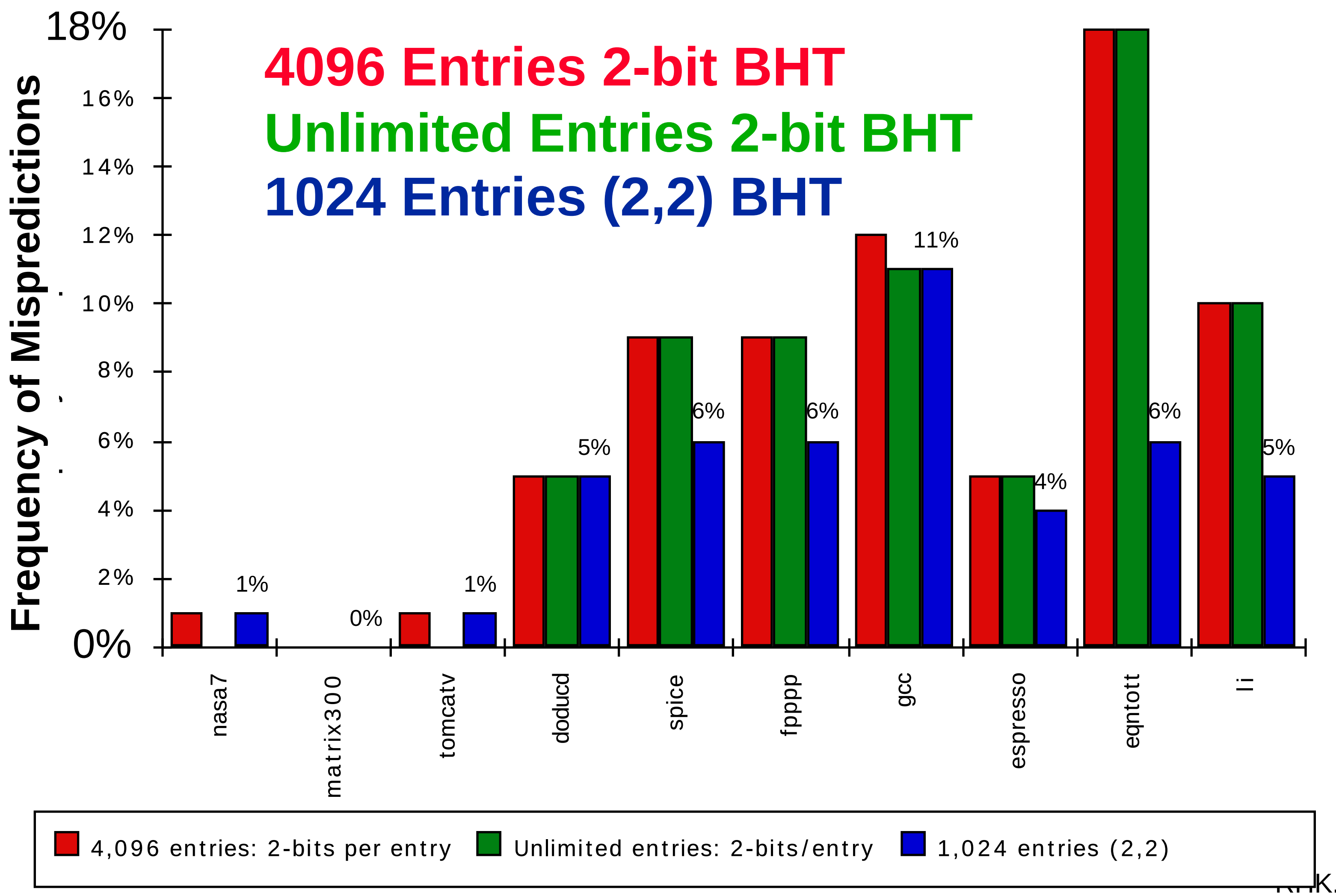
Idea: taken/not taken of recently executed branches is related to behavior of next branch (as well as the history of that branch behavior)

– Then behavior of recent branches selects between, say, four predictions of next branch, updating just that prediction



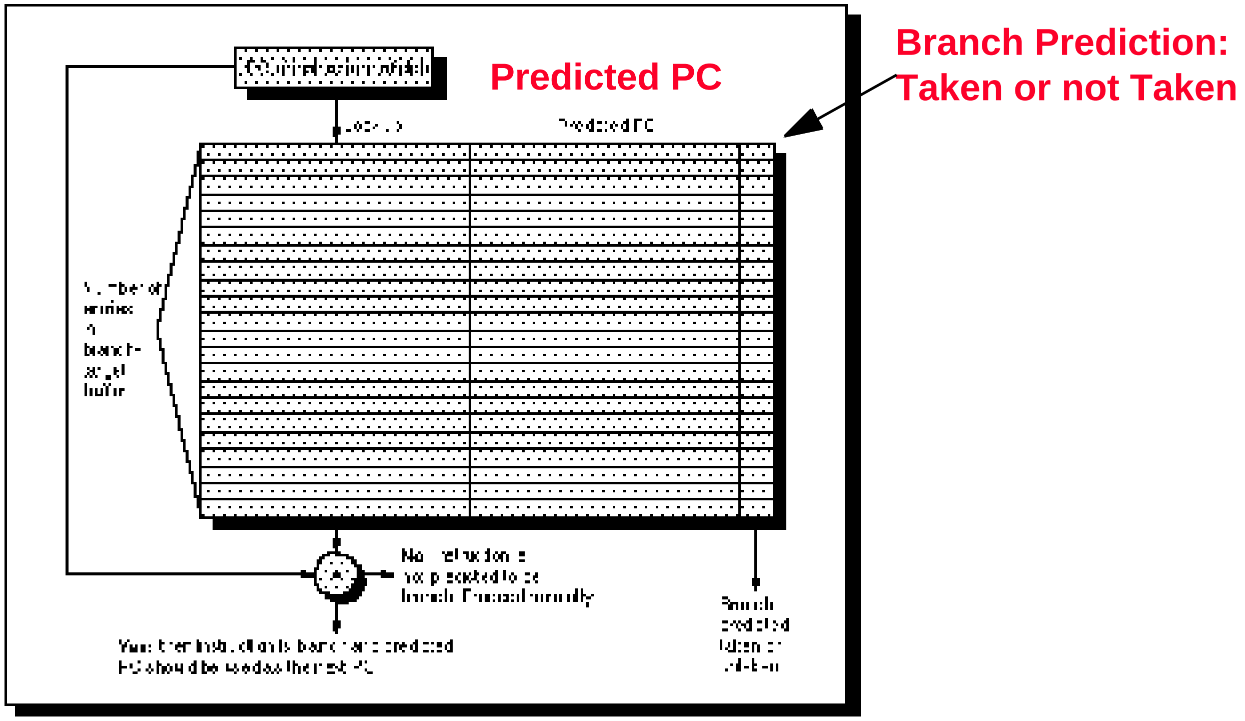
Accuracy of Different Schemes

(Figure 4.21, p. 272)



Need Address @ Same Time as Prediction

- **Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)**
  - **Note: must check for branch match now, since can't use wrong branch address** (Figure 4.22, p. 273)


PC of instruction to fetch
Lookup
Predicted PC
Predicted PC
Branch Prediction: Taken or not Taken
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

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- **Two variations**
- **Superscalar: varying no. instructions/cycle (1 to 8), scheduled by compiler or by HW (Tomasulo)**
  - **IBM PowerPC, Sun SuperSparc, DEC Alpha, HP 7100**
- **Very Long Instruction Words (VLIW): fixed number of instructions (16) scheduled by the compiler**
  - **Joint HP/Intel agreement in 1998?**

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- **Superscalar DLX: 2 instructions, 1 FP & 1 anything else**
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - More ports for FP registers to do FP load & FP op in a pair

Type	*PipeStages*						
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

- **1 cycle load delay expands to 3 instructions in SS**
  - instruction in right half can't use it, nor instructions in next slot

Unrolled Loop that Minimizes Stalls for Scalar

```
1 Loop: LD      F0,0(R1)
2       LD      F6,-8(R1)
3       LD      F10,-16(R1)
4       LD      F14,-24(R1)
5       ADDD    F4,F0,F2
6       ADDD    F8,F6,F2
7       ADDD    F12,F10,F2
8       ADDD    F16,F14,F2
9       SD      0(R1),F4
10      SD      -8(R1),F8
11      SD      -16(R1),F12
12      SUBI    R1,R1,#32
13      BNEZ    R1,LOOP
14      SD      8(R1),F16    ; 8-32 = -24
```

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

14 clock cycles, or 3.5 per iteration

Loop Unrolling in Superscalar

	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	LD F0,0(R1)		1
	LD F6,-8(R1)		2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12		8
	SD -24(R1),F16		9
	SUBI R1,R1,#40		10
	BNEZ R1,LOOP		11
	SD -32(R1),F20		12

- **Unrolled 5 times to avoid delays (+1 due to SS)**
- **12 clocks, or 2.4 clocks per iteration**

Dynamic Scheduling in Superscalar

- **Dependencies stop instruction issue**
- **Code compiler for scalar version will run poorly on SS**
  - **May want code to vary depending on how superscalar**
- **Simple approach: separate Tomasulo Control for separate reservation stations for Integer FU/Reg and for FP FU/Reg**

Dynamic Scheduling in Superscalar

- How to do instruction issue with two instructions and keep in-order instruction issue for Tomasulo?
  - Issue 2X Clock Rate, so that issue remains in order
  - Only FP loads might cause dependency between integer and FP issue:
    - Replace load reservation station with a load queue; operands must be read in the order they are fetched
    - Load checks addresses in Store Queue to avoid RAW violation
    - Store checks addresses in Load Queue to avoid WAR,WAW

Performance of Dynamic SS

Iteration no.	*Instructions*	*Issues*	*Executes*	*Writes result*
		clock-cycle number		
1	LD F0,0(R1)	1	2	4
1	ADDD F4,F0,F2	1	5	8
1	SD 0(R1),F4	2	9	
1	SUBI R1,R1,#8	3	4	5
1	BNEZ R1,LOOP	4	5	
2	LD F0,0(R1)	5	6	8
2	ADDD F4,F0,F2	5	9	12
2	SD 0(R1),F4	6	13	
2	SUBI R1,R1,#8	7	8	9
2	BNEZ R1,LOOP	8	9	

≈ 4 clocks per iteration

Branches, Decrements still take 1 clock cycle

Limits of Superscalar

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards
- If more instructions issue at same time, greater difficulty of decode and issue
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue
- VLIW: tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word can execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - » 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

Loop Unrolling in VLIW

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

- Unrolled 7 times to avoid delays
- 7 results in 9 clocks, or 1.3 clocks per iteration
- Need more registers in VLIW

Limits to Multi-Issue Machines

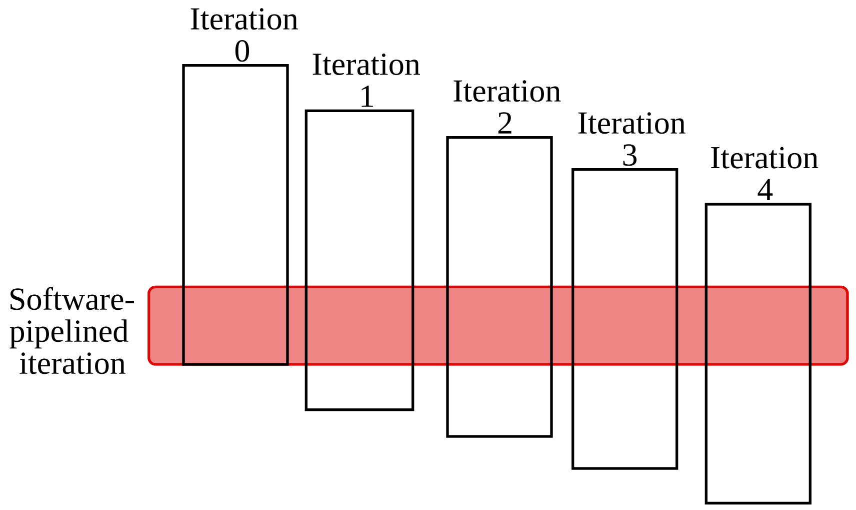
- Inherent limitations of ILP
  - 1 branch in 5 instructions => how to keep a 5-way VLIW busy?
  - Latencies of units => many operations must be scheduled
  - Need about Pipeline Depth x No. Functional Units of independent operations to keep machines busy
- Difficulties in building HW
  - Duplicate FUs to get parallel execution
  - Increase ports to Register File (VLIW example needs 6 read and 3 write for Int. Reg. & 6 read and 4 write for FP reg)
  - Increase ports to memory
  - Decoding SS and impact on clock rate, pipeline depth

Limits to Multi-Issue Machines

- **Limitations specific to either SS or VLIW implementation**
  - **Decode issue in SS**
  - **VLIW code size: unroll loops + wasted fields in VLIW**
  - **VLIW lock step => 1 hazard & all instructions stall**
  - **VLIW & binary compatibility is practical weakness**

Software Pipelining

- **Observation: if iterations from loops are independent, then can get 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)**



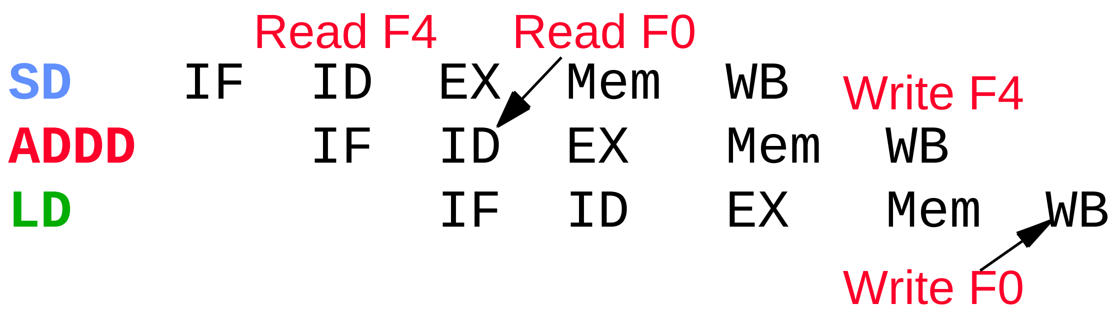
SW Pipelining Example

Before: Unrolled 3 times

```
1   LD    F0,0(R1)
2   ADDD  F4,F0,F2
3   SD    0(R1),F4
4   LD    F6,-8(R1)
5   ADDD  F8,F6,F2
6   SD    -8(R1),F8
7   LD    F10,-16(R1)
8   ADDD  F12,F10,F2
9   SD    -16(R1),F12
10  SUBI  R1,R1,#24
11  BNEZ  R1,LOOP
```

After: Software Pipelined

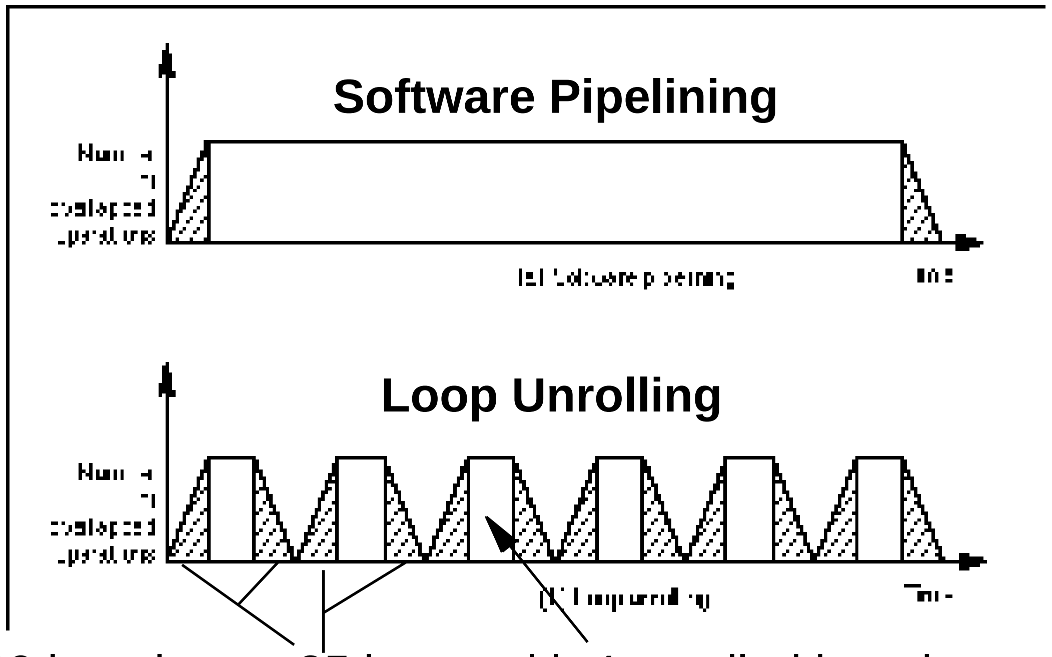
```
    LD    F0,0(R1)
    ADDD  F4,F0,F2
    LD    F0,-8(R1)
1   SD    0(R1),F4;     Stores M[i]
2   ADDD  F4,F0,F2;     Adds to M[i-1]
3   LD    F0,-16(R1);   loads M[i-2]
4   SUBI  R1,R1,#8
5   BNEZ  R1,LOOP
    SD    0(R1),F4
    ADDD  F4,F0,F2
    SD    -8(R1),F4
```



SW Pipelining Example

Symbolic Loop Unrolling

- *Less code space*
- Overhead paid only once
  vs. each iteration in loop unrolling



100 iterations = 25 loops with 4 unrolled iterations each

Summary

- **Branch Prediction**
  - Branch History Table: 2 bits for loop accuracy
  - Correlation: Recently executed branches correlated with next branch
  - Branch Target Buffer: include branch address & prediction
- **Superscalar and VLIW**
  - CPI < 1
  - Dynamic issue vs. Static issue
  - More instructions issue at same time, larger the penalty of hazards
- **SW Pipelining**
  - Symbolic Loop Unrolling to get most from pipeline with little code expansion, little overhead