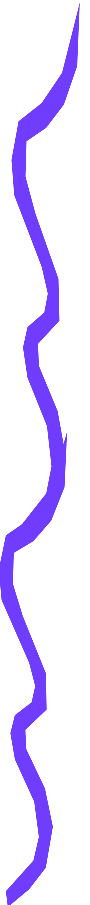
DEPT. OF ELECTRONICS ENGINEERING &
INST. OF ELECTRONICS
NCTU

5008: Computer Architecture

Chapter 2 – Instruction-Level Parallelism and Its Exploitation

Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- (Start) Tomasulo Algorithm
- Conclusion



DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Recall from Pipelining Review

- Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls
  - Ideal pipeline CPI: measure of the maximum performance attainable by the implementation
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Instruction-Level Parallelism

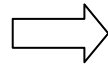
- The technique of increasing the ability of the processor to exploit parallelism among instructions.
  - To overlap the execution of instructions to improve performance
  - To reduce the impact of data and control hazards
- 2 approaches to exploit ILP
  - Rely on hardware to help discover and exploit the parallelism dynamically :
    - depend on the hardware to locate the parallelism
    - e.g., Pentium 4, AMD Opteron, IBM Power
  - Rely on software technology to find parallelism, statically at compile-time :
    - determine the parallelism at compiler time
    - e.g., Itanium 2

Instruction-Level Parallelism (ILP)

- Basic Block (BB) ILP is quite small :
  - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - average dynamic branch frequency 15% to 25%
    => 4 to 7 instructions execute between a pair of branches
  - Plus instructions in BB likely to depend on each other
- We must exploit ILP across multiple basic blocks

```
for (i=1; i<=1000, i=i+1)
  x[i] = x[i] + y[i]
```

Loop-level parallelism



```
x[1] = x[1] + y[1]
x[2] = x[2] + y[2]
…
x[1000]=x[1000]+y[1000]
```

  - Loop unrolling to exploit loop-level parallelism
    - By compiler, statistically
    - By hardware, dynamically
  - Vector instructions
    - The long latency of each vector instruction can be pipelined and operated in parallel

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Data Dependences and Parallelisms

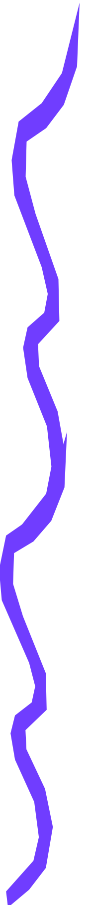
- If 2 instructions are parallel
  - they can execute simultaneously in a pipeline without causing any stalls (except the structural hazards)
  - their execution order can be swapped
- If 2 instructions are dependent
  - they must execute in order or partially overlapped.

- Exploit parallelism over instructions

➔ Determine dependences over instructions

3 Types of Data Dependences

- (True) Data dependences
- Name dependences
- Control dependences



Data Dependences

- A property of the programs.
- An instruction *j* is data dependent on instruction *i* if either of the following holds
  - instruction *i* produces a result that may be used by instruction *j*
  - instruction *j* is data dependent on instruction *k*, and instruction *k* is data dependent on instruction *I* (a chain of dependences)
- Data dependent instructions
  - indicates the possibility of a pipeline hazard
  - determines the program order in which results must be calculated
  - sets an upper bound on how much parallelism can possibly be exploited

Data Dependence Example

```
Loop:  L.D      F0, 0(R1)       ;F0=array element
       ADD.D    F4, F0, F2      ;add scalar in F2
       S.D      F4, 0(R1)       ;store result
       DADDUI   R1, R1, #-8     ;decrement pointer
       BNE      R1, R2, Loop    ;branch R1!=R2
```

- The arrows show the order that must be preserved for correct execution.
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped.

ILP and Data Dependencies

- HW/SW must preserve program order:
  order instructions would execute in if executed sequentially as determined by original source program
  - Dependences are a property of programs
- Presence of dependence indicates potential for a hazard, but actual hazard and length of any stall is property of the pipeline
- Importance of the data dependencies
  1) indicates the possibility of a hazard
  2) determines order in which results must be calculated
  3) sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program

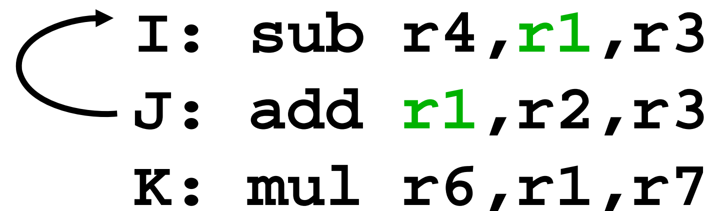
Overcome the Data Dependence

- Maintaining the dependence but avoiding a hazard
  - scheduling the code in HW/SW approach
- Eliminating a dependence by transforming the code
  - primary by software
- Dependence detection
  - by register names: simpler
  - by memory locations: more complicated
    - Two addresses may refer to the same location but look quite different (e.g. 100(R4), 20(R6) may be identical)
    - The effective address of a load/store may changed from instruction to instruction (20(R4), 20(R4) may be different)

Recall Data Hazards

- A hazard is created when a dependence between instructions closed enough
  - This affects the outcome of the program, since the operand access in wrong order
- The possible data hazards
  - RAW: the true data dependence
  - WAW: the output data dependence
  - WAR: the anti-dependence
  - RAR: this is not a hazard

- Name dependence: when 2 instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name; 2 versions of name dependence
- $Instr_J$ writes operand *before* $Instr_I$ reads it

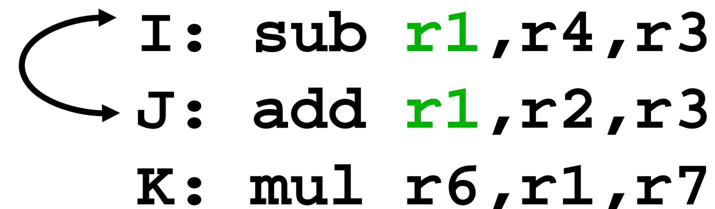


Called an "anti-dependence" by compiler writers.
This results from reuse of the name "r1"

- If anti-dependence caused a hazard in the pipeline, called a Write After Read (WAR) hazard

Name Dependence: Output dependence

- $Instr_J$ writes operand *before* $Instr_I$ writes it.



```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1"
- If anti-dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard
- Instructions involved in a name dependence can execute simultaneously if name used in instructions is changed so instructions do not conflict
  - Register renaming resolves name dependence for regs
  - Either by compiler or by HW

Register Renaming and WAW/WAR

- DIV.D F0, F2, F4
- ADD.D F6, F0, F8
- S.D F6, 0 (R1)
- SUB.D F8, F10, F14
- MUL.D F6, F10, F8

> - WAW: ADD.D/MUL.D
> - WAR: ADD.D/SUB.D, S.D/MUL.D
> - RAW: DIV.D/ADD.D, ADD.D/S.D SUB.D/MUL.D

- DIV.D F0, F2, F4
- ADD.D S, F0, F8
- S.D S, 0 (R1)
- SUB.D T, F10, F14
- MUL.D F6, F10, T

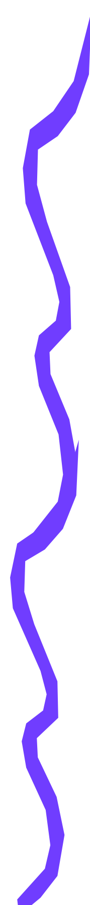
Register renaming result

Control Dependencies

- Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order

```
if p1 {
  S1;
};
if p2 {
  S2;
}
```

- S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.



DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Control Dependences

```
if p1 {
  s1
};
A
if p2 {
  s2
};
```

- Control dependence is preserved by two properties in a simple pipeline:
  - instruction execution in program order
  - control/branch hazard detection
    - ensure that an instruction that is control dependent on a branch is not executed until the branch direction is known
- Branch instructions (branches are conditional…)
  - An instruction that is control dependent on a branch cannot be moved before the branch
    - We cannot take an instruction from the then portion of an if statement and move it before the if statement
  - An instruction that is not control dependent on a branch cannot be moved after the branch
    - We cannot take a statement before the if statement and move it into the then portion

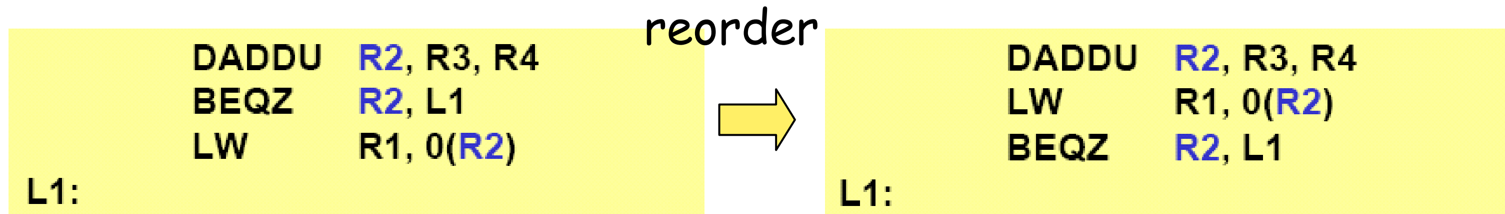
Control Dependence Ignored

- The method to preserve the control dependence
  - Be used in most simple pipeline CPUs
  - Simple but inefficient
- Control dependence is not the critical property that must be preserved
  - We may execute instruction that should not have been executed, thereby violating the control dependence, if we can do so without affecting the correctness of the program
- The two properties critical to program correctness are
  - The exception behavior
  - Data flow

Preserve the Exception Behavior

Control dependences

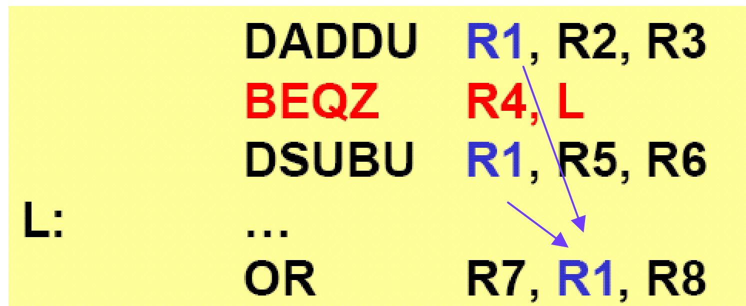
- Preserving exception behavior
  ⇒ any changes in instruction execution order must not change how exceptions are raised in program
  (⇒ no new exceptions)
- Example



  - LW, BEQZ : control dependence, but not data dependence
  - If interchange the order (still preserve the data dependence)
    - We must ignore the possible exception (memory protection due to LW instruction) when the branch is taken
      - Speculation technique should take care of this issue

Preserve Data Flow Behavior

- The data flow is the actual flow of data among instructions
- Branches make data flow dynamic
- Example



```
        DADDU   R1, R2, R3
        BEQZ    R4, L
        DSUBU   R1, R5, R6
L:      …
        OR      R7, R1, R8
```

  - R1 value depends on the branch is taken or not
  - DSUBU cannot be moved above the branch.
  - Speculation should take care this problem
    - Program order, that determines which predecessor will actually deliver a data value to the instruction, should be ensured by maintaining the control dependences

Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- (Start) Tomasulo Algorithm
- Conclusion

Software Techniques - Example

- This code, add a scalar to a vector:

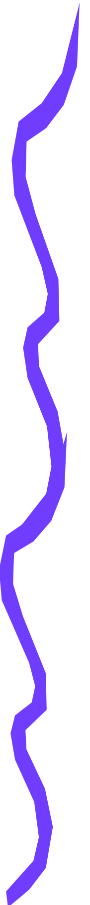
```
for (i=1000; i>0; i=i-1)
     x[i] = x[i] + s;
```

- Assume following latencies for all examples
  - Ignore delayed branch in these examples

Instruction producing result	*Instruction using result*	*Latency in cycles*	*stalls between in cycles*
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	1	1
Load double	Store double	1	0
Integer op	Integer op	1	0

```
for (i=1000; i>0; i=i-1)
            x[i] = x[i] + s;
```

- First translate into MIPS code:
  - To simplify, assume 8 is lowest address
  - R1 is initially the address of the element in the array with the highest address
  - F2 contains the scalar value s



```
Loop: L.D    F0,0(R1) ;F0=array element
      ADD.D  F4,F0,F2 ;add scalar from F2
      S.D    0(R1),F4 ;store result
      DADDUI R1,R1,-8 ;decrement pointer 8B (DW)
      BNEZ   R1,Loop  ;branch R1!=zero
```

Where are the hazards?

FP Loop Showing Stalls

```
1 Loop: L.D    F0,0(R1) ;F0=array element
2       stall
3       ADD.D  F4,F0,F2 ;add scalar in F2
4       stall
5       stall
6       S.D    0(R1),F4 ;store result
7       DADDUI R1,R1,-8 ;decrement pointer 8B (DW)
8       stall           ;assumes can't forward to branch
9       BNEZ   R1,Loop  ;branch R1!=zero
```

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

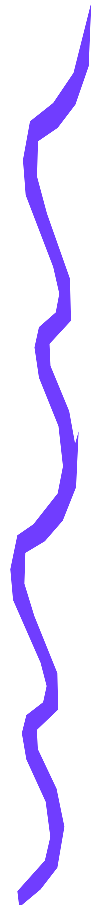
- 9 clock cycles: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

```
1 Loop: L.D    F0,0(R1)
2       DADDUI R1,R1,-8
3       ADD.D  F4,F0,F2
4       stall
5       stall
6       S.D    8(R1),F4;altered offset when move DSUBUI
7       BNEZ   R1,Loop
```

Swap `DADDUI` and `S.D` by changing address of `S.D`

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



7 clock cycles, but just 3 for execution (`L.D`, `ADD.D`, `S.D`), 4 for loop overhead; How make faster?

Unroll Loop Four Times (straightforward way)

```
1  Loop:L.D     F0,0(R1)
3       ADD.D   F4,F0,F2
6       S.D     0(R1),F4      ;drop DSUBUI & BNEZ
7       L.D     F6,-8(R1)
9       ADD.D   F8,F6,F2
12      S.D     -8(R1),F8     ;drop DSUBUI & BNEZ
13      L.D     F10,-16(R1)
15      ADD.D   F12,F10,F2
18      S.D     -16(R1),F12   ;drop DSUBUI & BNEZ
19      L.D     F14,-24(R1)
21      ADD.D   F16,F14,F2
24      S.D     -24(R1),F16
25      DADDUI  R1,R1,#-32    ;alter to 4*8
27      BNEZ    R1,LOOP
```

1 cycle stall

2 cycles stall

Rewrite loop to minimize stalls?

27 clock cycles, or 6.75 per iteration

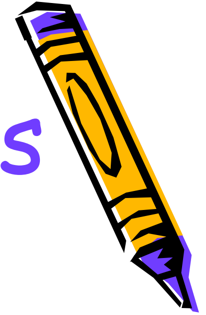
(Assumes R1 is multiple of 4)

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

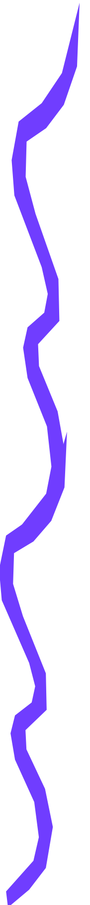
Unrolled Loop Detail

- Do not usually know upper bound of loop
- Suppose it is n, and we would like to unroll the loop to make k copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
  - 1st executes (n mod k) times and has a body that is the original loop
  - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
- For large values of n, most of the execution time will be spent in the unrolled loop

Unrolled Loop That Minimizes Stalls



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



14 clock cycles, or 3.5 per iteration

5 Loop Unrolling Decisions

- Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:

1. Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code)
2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations
3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code
4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent
   - Transformation requires analyzing memory addresses and finding that they do not refer to the same address
5. Schedule the code, preserving any dependences needed to yield the same result as the original code

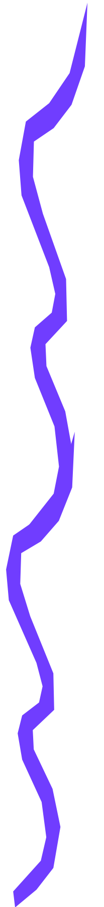
3 Limits to Loop Unrolling

1. Decrease in amount of overhead amortized with each extra unrolling
   - Amdahl's Law
2. Growth in code size
   - For larger loops, concern it increases the instruction cache miss rate
3. Register pressure (compiler limitation): potential shortfall in registers created by aggressive unrolling and scheduling
   - If not be possible to allocate all live values to registers, may lose some or all of its advantage

- Loop unrolling reduces impact of branches on pipeline; another way is branch prediction

Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- (Start) Tomasulo Algorithm
- Conclusion



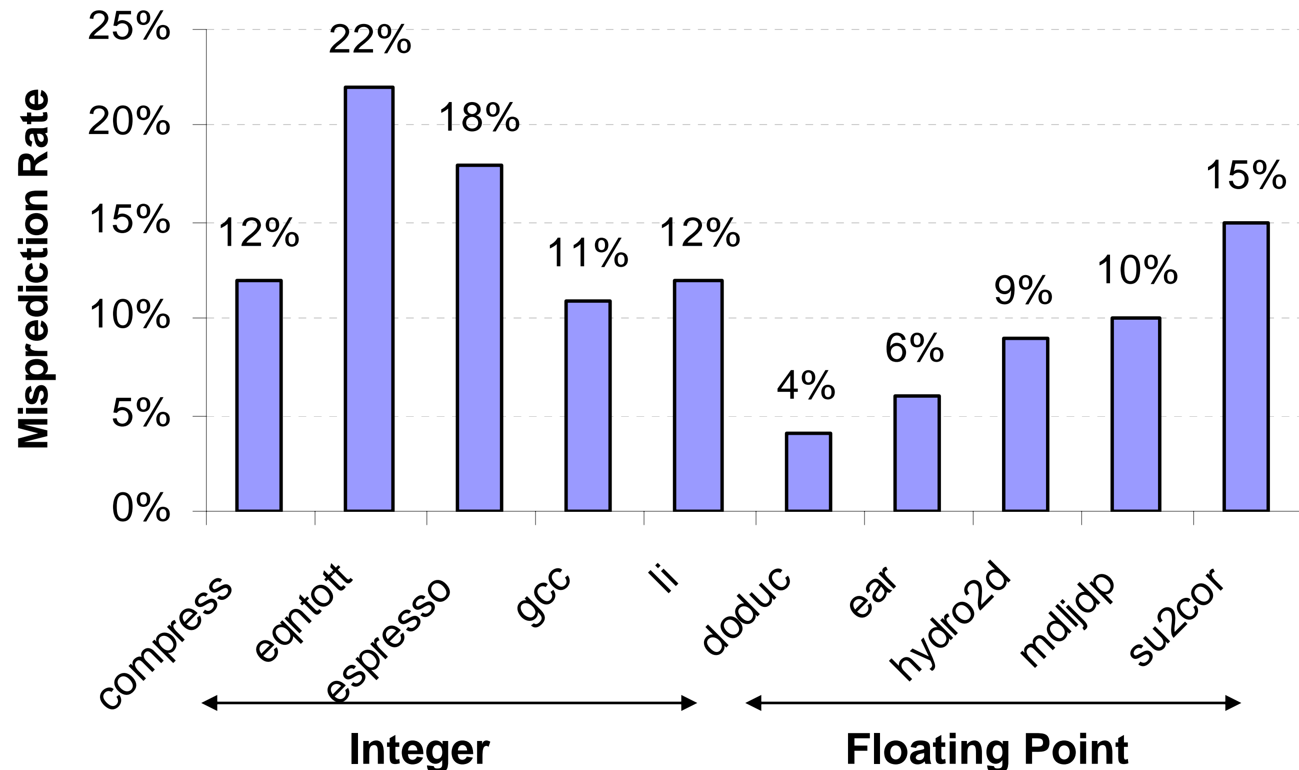
Control Hazard Avoidance

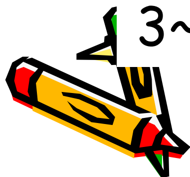
- Consider Effects of Increasing the ILP
  - Control dependencies rapidly become the limiting factor
  - They tend to not get optimized by the compiler
    - Higher branch frequencies result
    - Plus multiple issue (more than one instructions/sec) → more control instructions per sec.
  - Control stall penalties will go up as machines go faster
    - Amdahl's Law in action - again
- Branch Prediction: helps if can be done for reasonable cost
  - Static by compiler: appendix A
    e.g. predict not taken, delay branch
  - Dynamic by HW: this section

Static Branch Prediction

- Lecture 3 showed scheduling code around delayed branch
- To reorder code around branches, need to predict branch statically when compile
- Simplest scheme is to predict a branch as taken
  - Average misprediction = untaken branch frequency = 34% SPEC

•More accurate scheme predicts branches using profile information collected from earlier runs, and modify prediction based on last run: 3~24%





Dynamic Branch Prediction

- Why does prediction work?
  - Underlying algorithm has regularities
  - Data that is being operated on has regularities
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems
- Is dynamic branch prediction better than static branch prediction?
  - Seems to be
  - There are a small number of important branches in programs which have dynamic behavior

Dynamic Branch Prediction

- The predictor will depend on the behavior of the branch at run time
- Goals:
  - allow the processor to resolve the outcome of a branch early, prevent control dependences from causing stalls
- Effectiveness of a branch prediction scheme depends not only on the accuracy but also on the cost of a branch
  - BP_Performance = f (accuracy, cost of misprediction)
- Branch History Table (BHT)
  - Lower bits of PC address index table of 1-bit values
    - No "precise" address check – just match the lower bits
  - Says whether or not branch taken last time

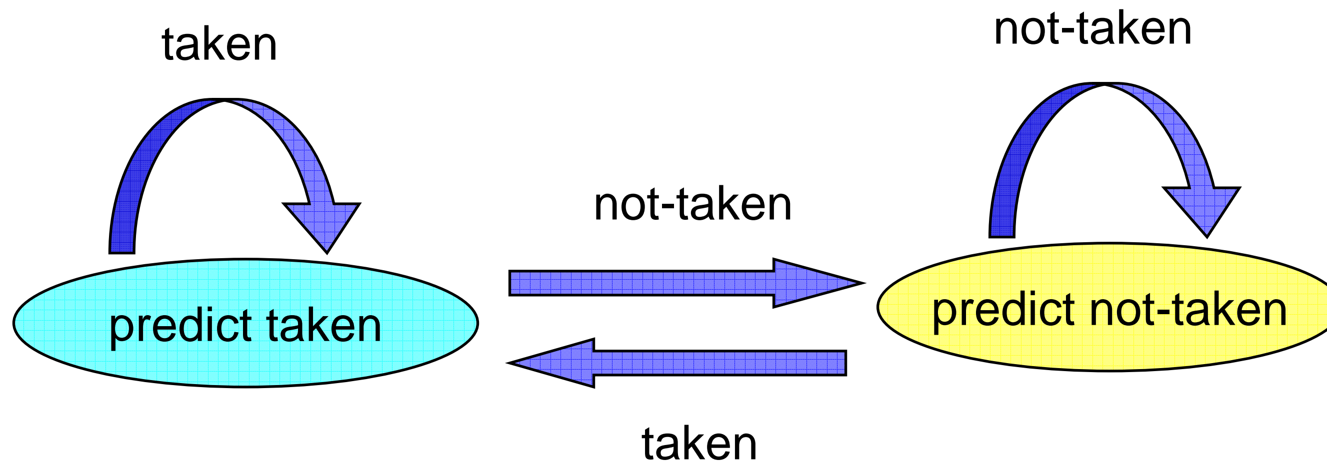
BHT Prediction

Useful only for the target address is known before CC is decided

If two branch instructions with the same lower bits…

Dynamic Hardware Prediction

One-bit prediction scheme



Problem: Loop case incorrect predict twice, even if the branch is almost always taken

```
LOOP:    LOAD    R1, 100(R2)
         MUL     R6, R6, R1
         SUBI    R2, R2, #4
         BNEZ    R2, LOOP
```

The steady-state prediction behavior will mispredict on the first and last loop iterations

Problem with the Simple BHT

clear benefit is that it's cheap and understandable

- Aliasing
  - All branches with the same index (lower) bits reference same BHT entry
    - Hence they mutually predict each other
    - No guarantee that a prediction is right. But it may not matter anyway
  - Avoidance
    - Make the table bigger - OK since it's only a single bit-vector
    - This is a common cache improvement strategy as well
      - Other cache strategies may also apply
- Consider how this works for loops
  - Always mispredict twice for every loop
    - One is unavoidable since the exit is always a surprise
    - However previous exit will always cause a mis-prediction the first try of every new loop entry

N-bit Predictors

idea: improve on the loop entry problem

- 2-bit counter implies 4 states
  - Statistically 2 bits gets most of the advantage

Compiler could hint 11 init. on loop branches or it will go to 11 anyway in the 4th iteration

A prediction must miss twice before it is changed

Only the loop exit causes a mispredict

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

2-Bit Predictor

- The branch prediction buffer (BPB) is a small special cache
  - accessed with the instruction address during the IF stage
  - 2-bit attached to each block in instruction cache and fetched with the instruction
  - No help for 5-stage pipeline processor
    - since in the ID stage, both whether the branch is taken and what the target of the branch is at roughly the same time

BHT Accuracy

4K of BPB with 2-bit entries misprediction rates on SPEC89@IBM Power

- Mispredict because either:
  - Wrong guess for that branch (accuracy)
  - Got branch history of wrong branch when index the table (size)

SPEC89 benchmarks

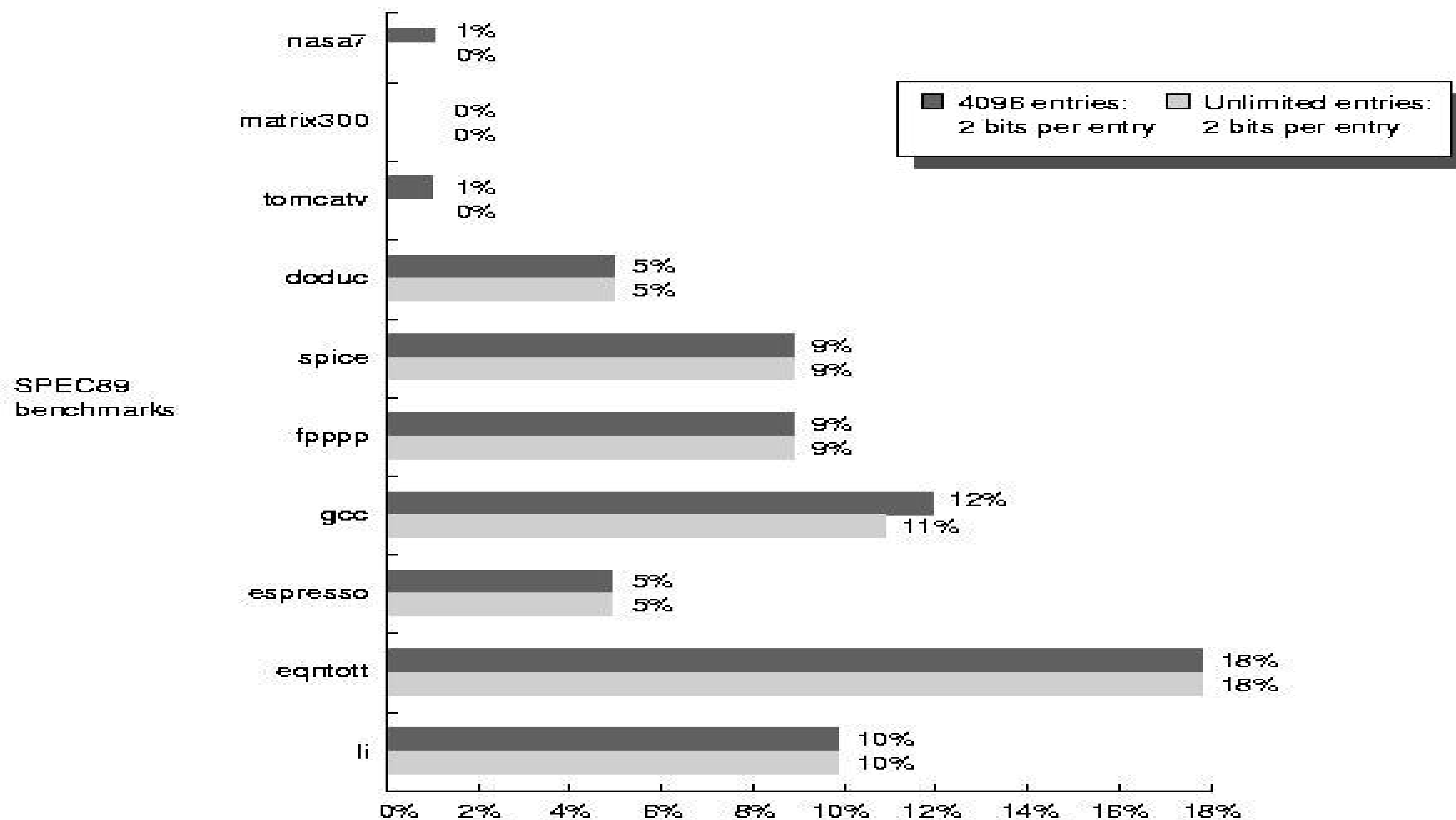
Benchmark	Frequency of mispredictions
nasa7	1%
matrix300	0%
tomcatv	1%
doduc	5%
spice	9%
fpppp	9%
gcc	12%
espresso	5%
eqntott	18%
li	10%

0% 2% 4% 6% 8% 10% 12% 14% 16% 18%

Frequency of mispredictions

To Increase the BHT Size

- 4096 about as good as infinite table
- The hit rate of the buffer is clearly not the limiting factor for an enough-large BHT size



The worst case for the 2-bit predictor

C code	MIPS code

if (aa==2)
  aa=0;
if (bb==2)
  bb=0;
if (aa != bb) {
	DSUBUI R3, R1, #2
	BNEZ R3, L1 ;branch b1(aa!=2)
	DADDD R1, R0, R0 ;aa=0
	L1: DSUBUI R3, R2, #2
	BNEZ R3, L2 ;branch b2(bb!=2)
	DADDD R2, R0, R0 ;bb=0
	L2: DSUBU R3, R1, R2
	BEQZ R3, L3 ;branch b3(aa==bb)
	
	aa and bb are assigned to R1 and R2
	
	if the first 2 untaken then the 3rd will always be taken

Improve Prediction Strategy By Correlating Branches

- Consider the worst case for the 2-bit predictor

```
if (aa==2) then aa=0;
if (bb==2) then bb=0;
if (aa != bb) then whatever
```

if the first 2 fail then the 3rd will always be taken

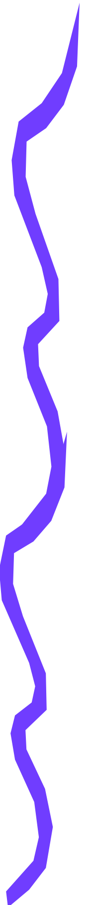
  - single level predictors can never get this case

- Correlating or 2-level predictors
  - The predictor uses the behavior of other branch(es) to make a prediction
  - Correlation = what happened on the last branch
  - Predictor = which way to go

Correlating Branches

Two-level predictors

- Hypothesis: recently executed branches are correlated
- Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper branch history table
- In general, (m,n) predictor means record last m branches to select between 2^m history tables each with n-bit counters
  - Old 2-bit BHT is then a (0,2) predictor
- Global Branch History: m-bit shift register keeping T/NT status of last m branches
- Each entry in table has m n-bit predictors



Generic (m,n) BHT

- (m,n) predictor
  - Use the behavior of the last m branches to choose from 2^m branch predictors, each of which is an n-bit predictor for a single branch
  - Total bits for the (m, n) BHT prediction buffer:

$$Total_memory_bits = 2^m \times n \times 2^p$$

  - p bits of buffer index = 2^p bit BHT
  - 2^m banks of memory selected by the global branch history (which is just a shift register) - e.g. a column address
  - Use p bits of the branch address to select row
  - Get the n predictor bits in the entry to make the decision

(2,2) Predictor Implementation

4 banks = each with 32 2-bit predictor entries

p=5
m=2
n=2

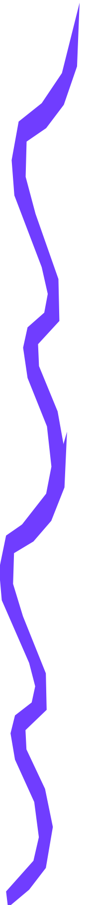
Note: orthogonal address model speeds decode common use is in DRAMS

Example of Correlating Branch Predictors

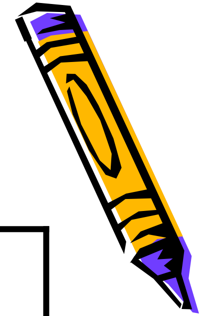
```
if (d==0)
    d = 1;
if (d==1)
    …
```

```
        BNEZ   R1, L1        ;branch b1 (d!=0)
        DAAIU  R1, R0, #1    ;d==0, so d=1
L1:     DAAIU  R3, R1, #-1
        BNEZ   R3, L2        ;branch b2 (d!=1)
…
L2:
```

d is assigned to R1



Example of Correlating Branch Predictors (Cont.)



initial value of d	d==0?	b1	value of d before b2	d==1?	b2
0	YES	not taken	1	YES	not taken
1	NO	taken	1	YES	not taken
2	NO	taken	2	NO	taken

1-bit predictor initialized to NT

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

All the branches are mispredicted !!!

Example of Correlating Branch Predictors (Cont.)

Prediction bits	Prediction if last branch not taken	Prediction if last branch taken
NT/NT	NT	NT
NT/T	NT	T
T/NT	T	NT
T/T	T	T

(1,1) predictor

Use 1-bit correlation + 1-bit prediction with initialized to NT/NT

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS

NCTU

Comparison: Accuracy of Different 2-bit Predictors

SPEC89 benchmarks	4096 entries: 2 bits per entry (8K bits)	Unlimited entries: 2 bits per entry	1024 entries: (2,2) (8K bits)
nasa7	1%	0%	1%
matrix300	0%	0%	0%
tomcatv	1%	0%	1%
doduc	5%	5%	5%
spice	9%	9%	5%
fpppp	9%	9%	5%
gcc	12%	11%	11%
espresso	5%	5%	4%
eqntott	18%	18%	6%
li	10%	10%	5%

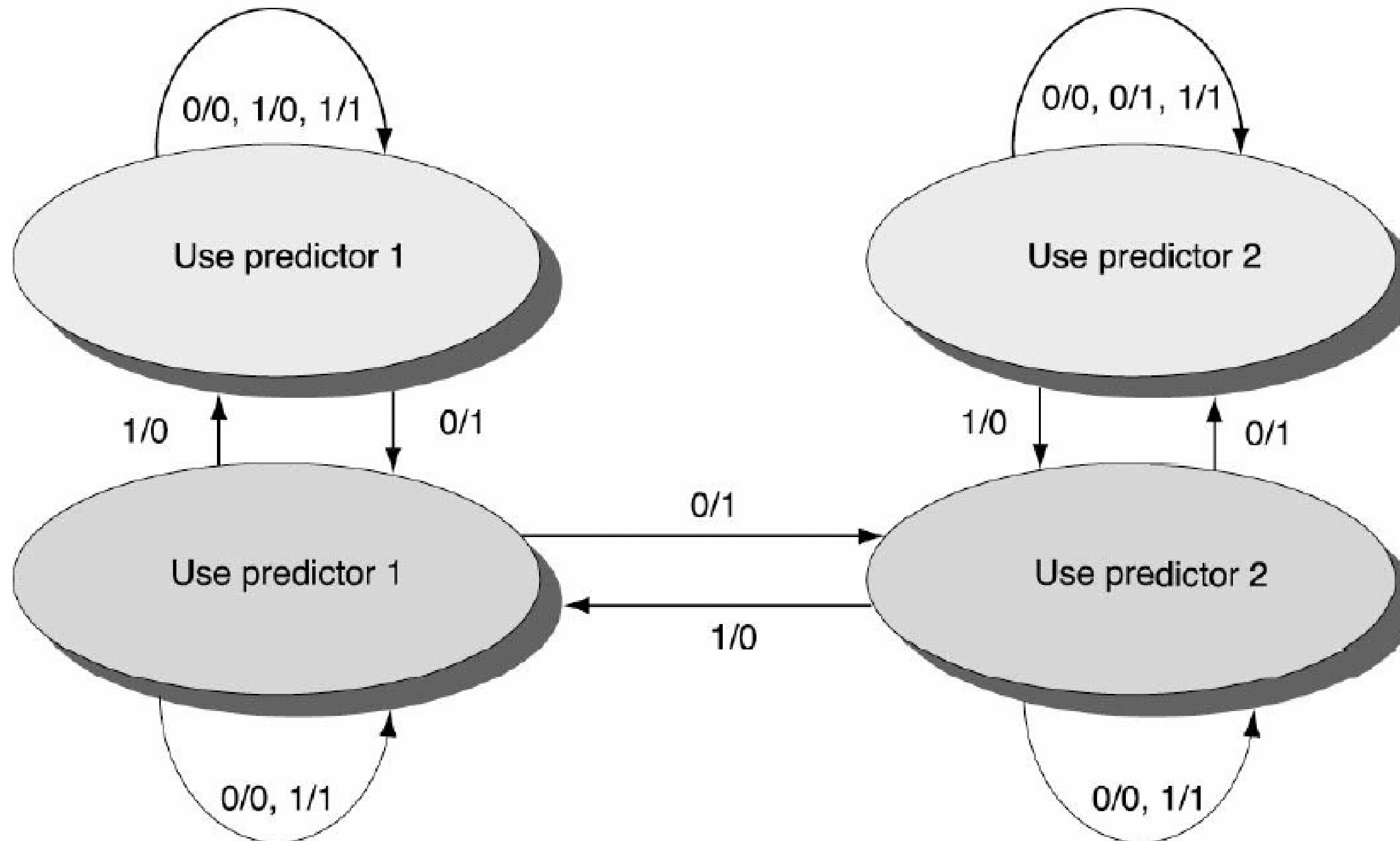
Frequency of mispredictions (0% – 18%)

Tournament Predictors

The most popular one

- Recall that the correlator is just a local predictor
- Adaptively combine local and global predictors
  - Multiple predictors
    - One based on global information: Results of recently executed m branches
    - One based on local information: Results of past executions of the current branch instruction
  - Selector to choose which predictors to use
    - E.g.: 2-bit saturating counter, incremented whenever the "predicted" predictor is correct and the other predictor is incorrect, and it is decremented in the reverse situation
- Advantage
  - Ability to select the right predictor for the right branch

State Transition Diagram for A Tournament Predictor



Misprediction Rate Comparison

Conditional branch misprediction rate

8%
7%
6%
5%
4%
3%
2%
1%
0%

Local 2-bit predictors

Correlating predictors

Tournament predictors

0 32 64 96 128 160 192 224 256 288 320 352 384 416 448 480 512

Total predictor size

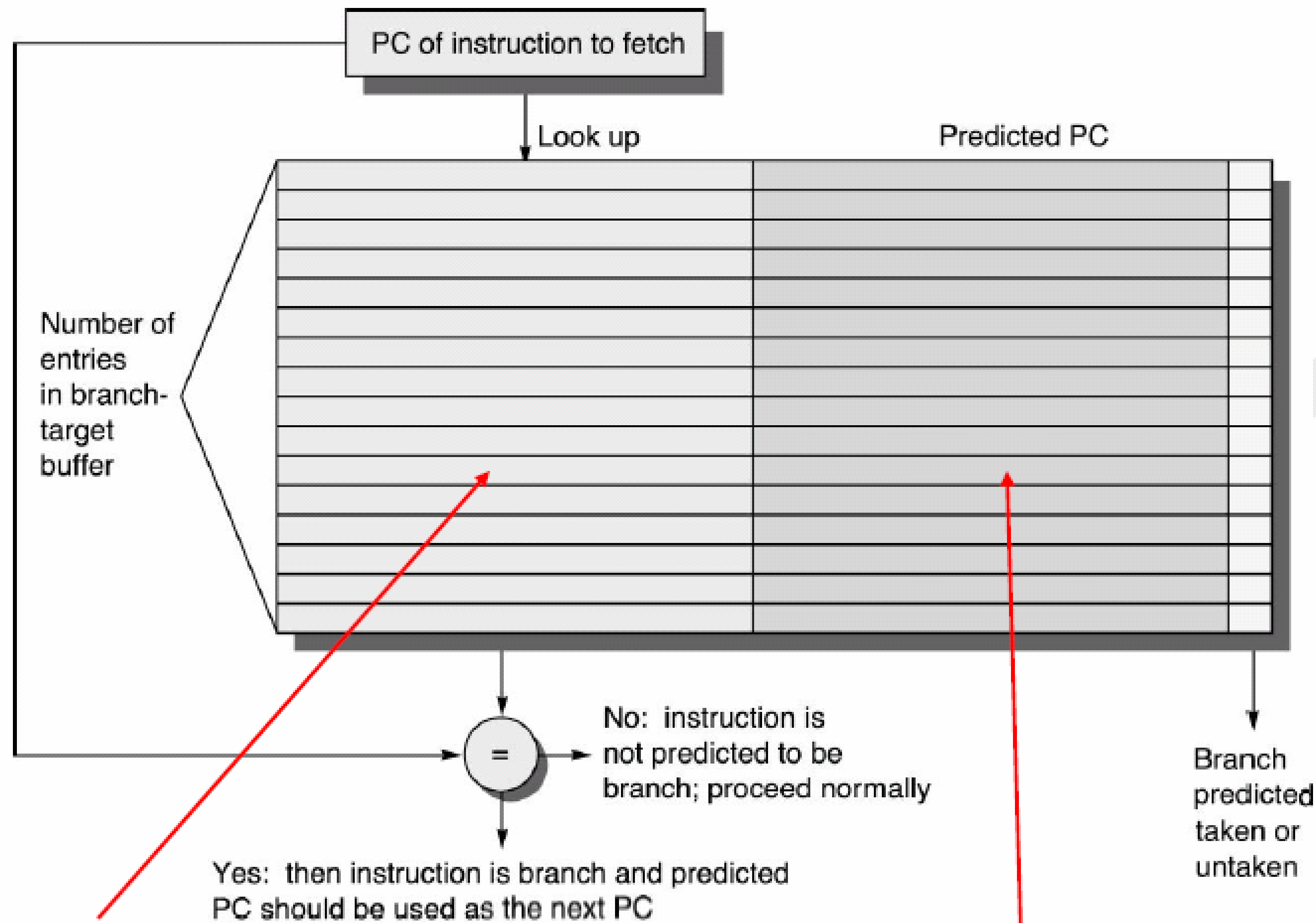
High-Performance Instruction Delivery

- For a multiple issue processor, predicting branches well is not enough
- Deliver a high-bandwidth instruction stream is necessary
  (e.g., 4~8 instructions/cycle)
  - Branch target buffer
  - Integrated instruction fetch unit
  - Indirect branch by predicting return address

Branch Target Buffer/Cache

- To reduce the branch penalty from 1 cycle to 0
  - Need to know what the address is at the end of IF
  - But the instruction is not even decoded yet
  - So use the instruction address rather than wait for decode
    - If prediction works then penalty goes to 0!
- BTB Idea -- Cache to store taken branches (no need to store untaken)
  - *Access the BTB during IF stage*
  - Match tag is instruction address → compare with current PC
  - Data field is the predicted PC
- May want to add predictor field
  - To avoid the mispredict twice on every loop phenomenon
  - Adds complexity since we now have to track untaken branches as well

BTB -- Illustration



Store predicted-taken branches only

Full size (32-bit)
No aliasing allowed

Target PCs for predicted-taken branches

Flowchart for BTB

IF

PC to memory and BTB

found in BTB?

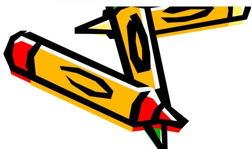
- NO → (to ID stage) taken — *A branch?*
  - NO → Normal Execution
  - YES → Enter branch addr and target PC into BTB
- YES → predicted PC to memory and BTB — Send out the predicted PC

ID

taken

- NO → Mispredict - kill fetched instruction restart fetch at other target, delete entry from BTB
- YES → Prediction correct, continue with no penalty

EX



DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Penalties Using this Approach for 5-Stage MIPS

Instruction in buffer	Prediction	Actual Branch	Penalty Cycles
Yes	Taken	Taken	0
Yes	Taken	Not Taken	2
No		Taken	2
No		Not Taken	0

Note:

- **Predict_wrong = 1 CC to update BTB + 1 CC to restart fetching**
- **Not found and taken = 2CC to update BTB**

Note:

- **For complex pipeline design, the penalties may be higher**

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Example

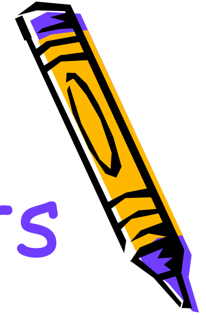
- Given prediction accuracy (for inst. in buffer): 90%
- Given hit rate in buffer (for branches predicted token): 90%
- Assume 60% of the branches are taken
- Determine the total branch penalty=?

Solution

- Probability (branch in buffer, but actually not taken) = percent buffer hit rate × percent incorrect prediction = 90% × 10% = 0.09
- Probability (branch not in buffer, but actually taken) = 10%
- Hence, we have 2 cycles × (0.09+0.1) = 0.38 cycles

Comparing the delay branch with the penalty = 0.5 cycles/branch

Integrated Instruction Fetch Units



- Consider the fetch unit as a separate autonomous unit, not a pipeline stage
- Functions for the integrated instruction fetch unit
  - Branch prediction
  - Prefetch
    - To deliver multiple instructions per cycle
  - Instruction memory access and buffering
    - may require accessing multiple cache lines
    - prefetch may hide the latency for memory access
    - buffering may be necessary

Return Address Predictor

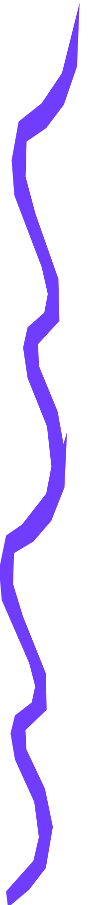
- Indirect jump – jumps whose destination address varies at run time
  - indirect procedure call, select or case, procedure return
  - SPEC89 benchmarks: 85% of indirect jumps are procedure returns
- Accuracy of BTB for procedure returns are low
  - if procedure is called from many places, and the calls from one place are not clustered in time
- Use a small buffer of return addresses operating as a stack
  - Cache the most recent return addresses
  - Push a return address at a call, and pop one off at a return
  - If the cache is sufficient large (max call depth) → prefect

Dynamic Branch Prediction Summary

- Branch prediction scheme are limited by
  - Prediction accuracy
  - Mis-prediction penalty
- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch
- Tournament predictors take insight to next level, by using multiple predictors
  - usually one based on global information and one based on local information, and combining them with a selector
  - In 2006, tournament predictors using ≈ 30K bits are in processors like the Power5 and Pentium 4
- Branch Target Buffer: include branch address & prediction
- Reduce penalty further by fetching instructions from both the predicted and unpredicted direction
  - Require dual-ported memory, interleaved cache → HW cost
  - Caching addresses or instructions from multiple path in BTB

Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- (Start) Tomasulo Algorithm
- Conclusion



Overcome Data Hazards

- For a simple statically scheduled pipeline
  - In-order instruction issue and execution
  - fetch an instruction and issue it in program order
  - if there is a data dependence that cannot be hidden (e.g. forwarding logic), then the hazard detection hardware stalls the pipeline
  - No new instructions are fetched or issued until the dependence is cleared.
  - Minimize stalls by software to separate dependent instructions so that they will not lead to hazards

Overcome Data Hazards

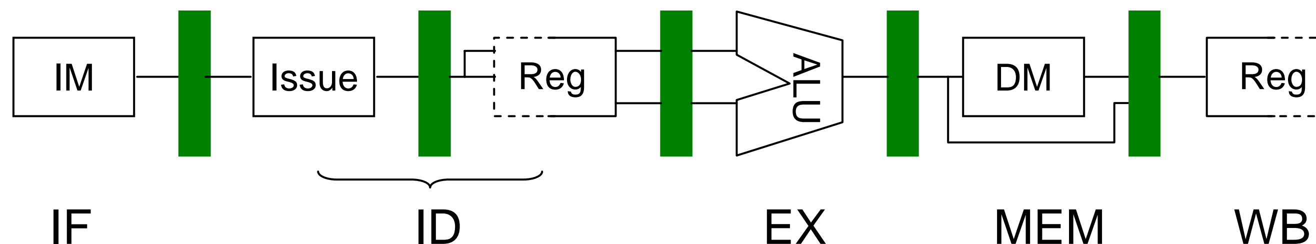
- For a dynamically scheduling
  - the hardware rearranges the instruction execution to reduce the stalls while maintaining data flow and exception behavior
  - Pros:
    - handling some cases when dependences are unknown at compiler time (e.g. memory reference)
    - simplify the compiler
    - (Perhaps most importantly) allow code compiled with one pipeline run on a different pipeline
    - will explore hardware speculation
  - Cons:
    - a cost of significant increase in hardware complexity

Remarks

- A dynamically scheduled processor cannot change the data flow
  - It tries to avoid stalling when dependences
- A static pipeline scheduling by compiler
  - It tries to minimize stalls by separating dependent instructions far away from the other such that they will not lead to hazards.

Dynamic Scheduling: OOO

- In classic 5-stage pipeline, both structural and data hazards could be checked during ID stage
  - When an instruction could execute without hazards, it was issued from ID knowing that all data hazards had been resolved.
- Let separate the ID stage into two parts
  - Issue:
    - Decode, check for structural hazard in the manner of in-order issue
  - Read Operands:
    - Wait until no data hazards, then read operands
- Out-of-order (OOO) execution
  - It allows the instruction to begin execution as soon as its data operand is available
  - It implies out-of-order completion
  - It may introduce WAR, WAW hazards



OOO Example

- In-order issue, but allow out-of-order execution (and thus out-of-order completion)

Example 1

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

SUB.D has dependence with neither DIV.D nor ADD.D

However, it cannot execute if **out-of-order execution** is not allowed.

Performance limitation due to hazard...

Example 2

```
DIV.D   F0, F2, F4
ADD.D   F6, F0, F8        ; stalled
SUB.D   F8, F10, F14
MUL.D   F6, F10, F8
```

However, if out-of-order execution is allowed, **WAR** or **WAW** hazards could arise

Eliminating WAR and WAW hazards is essential to out-of-order execution → ***Register Renaming***

WAR & WAW May Arise When Dynamic Scheduling

- Both WAW and WAR hazards can be solved by
  - Scoreboard (Appendix A, used in CDC6600 first) and
  - Tomasulo approach (used in IBM 360/91 Floating-point Unit)

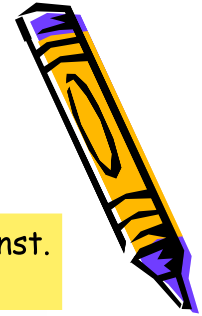
Scoreboard

- Idea:
  - to maintain IPC=1 by executing an instruction as early as possible
  - when stalled, other instructions can be issued and executed if they do not depend on any active or stalled instruction
- The scoreboard takes full responsibility for instruction issue and execution, including hazard detection (centralized control)
- 3 parts to the scoreboard
  - Instruction status: indicating the pipeline stage of the instruction
  - Functional unit status: 9 fields
  - Register result status : which FU will write the result to register

Functional Unit Status in Scoreboard

- 9 fields:
  - Busy
  - Op – operation performing (e.g. add, sub)
  - Fi – destination register
  - Fj, Fk – source register
  - Qj, Qk – FU that produces Fj, Fk
  - Rj, Rk – flags of Fj, Fk to indicate being read or not (set to No after operands are read)

Scoreboard Example (1/3)



The time that 2nd inst. is read to write

Instruction		Issue	Read operands	Execution complete	Write result
L.D	F6,34(R2)	√	√	√	√
L.D	F2,45(R3)	√	√	√	Read to write
MUL.D	F0,F2,F4	√			
SUB.D	F8,F6,F2	√			
DIV.D	F10,F0,F6	√			
ADD.D	F6,F8,F2				

Issued but stalled for waiting operands

Each instr. has an entry

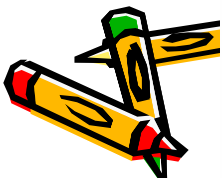
Functional unit status

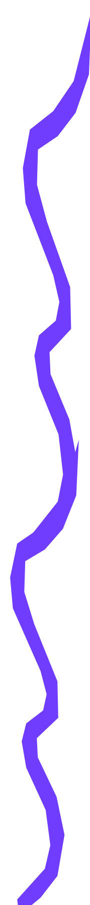
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	R3				No	
Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Each FU has an entry

Register result status

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	Integer			Add	Divide			





Scoreboard Example (2/3)

The time before the MUL.D goes to write the result

Add : 2 cycles, Multiply: 10 cycles, Divide: 40 cycles

Instruction		Issue	Read operands	Execution complete	Write result
		Instruction status			
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	√	√	√	

Out-of-order completion

WAR

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 Example (3/3)

The time that DIV.D goes to write the result

Instruction		Instruction status			
		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	No								
Mult2	No								
Add	No								
Divide	Yes	Div	F10	F0	F6			No	No

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

Remarks

- The scoreboard identifies what requires for each instruction to advance and bookkeeping action necessary when the instruction does advance
- The scoreboard records operand specifier information, such as register number
- Costs and benefits of scoreboard
  - The amount of parallelism available among the instructions
    - Whether independent instructions can be found ?
  - The number of scoreboard entries
    - The instrs. window the pipeline can look for independent instr.
  - The number and types of FUs
  - The presence of anti-dependence and output dependence
- Recently, the dynamic scheduling is motivated by attempts to issue more instructions per clock and by speculation

Tomasulo's Approach

- Goal: High Performance without special compilers
- The original idea is for IBM 360/91; to overcome
  - limited compiler scheduling (only 4 double-precision FP registers)
  - reduce memory accesses and FP delays
- Why study 1966 computer?
  - lead to Alpha 21264, HP 8000, MIPS 10000, Pentium II, PowerPC 604, …
- Key ideas
  - Track data dependences to allow execution as soon as operands are available → minimize RAW hazards
  - Register renaming → avoid WAR and WAW hazards

Remark

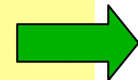
- To solve RAW hazard
  - The RAW problem can be avoided for an instruction only when its operands are available
- To solve WAR and WAW hazards
  - These problems arise from name dependence
  - By renaming all destination registers, including those with a pending read or write for an earlier instruction, the WAR and WAW can be avoided for out-of-order completion instructions.

Register Renaming Example

DIV.D	F0, F2, F4
ADD.D	F6, F0, F8
S.D	F6, 0(R1)
SUB.D	F8, F10, F14
MUL.D	F6, F10, F8

3 true dependences
2 antidependences
1 output dependence

Register



Renaming

DIV.D	F0, F2, F4
ADD.D	S, F0, F8
S.D	S, 0(R1)
SUB.D	T, F10, F14
MUL.D	F6, F10, T

3 true dependences
0 antidependence
0 output dependence

- Renaming process can also be done by compiler
- Tamasulo's algorithm can handle renaming across branches
- In Tomasulo's algorithm, register renaming is provided by the reservation station (RS)

Reservation Station (RS)

- To buffer the operands, as soon as it is available, waiting to issue
  - In order to eliminate the need to get the operand from a register (similar to forwarding)
- Pending instructions designate the reservation station (avoid WAR)
  - Operands of pending instructions are provided from RS rather than from the RF
  - Pending operands are renamed to the name of reservation stations
- When successive writes to a register overlap in execution, only last one is actually used to update the register (avoid WAW)

RSs vs. RFs

- Reservation stations serve as extra virtual registers
- Hazard detection and execution control are distributed (distributed control)
  - The information of RSs at each FU determine when an instruction can begin execution at that FU
- Results are passed directly to FUs from the RSs where they are buffered, rather than going through the RFs
  - This can be done with a common result bus that allows all FUs waiting for the operand to be loaded simultaneously

Basic Structure of A Tomasulo-Based MIPS Processor

From instruction unit

Instruction queue

FP registers

Load-store operations

Floating-point operations

Operand buses

Address unit

Store buffers

Load buffers

Operation bus

Virtual registers

3
2
1

Reservation stations

2
1

Data

Address

Memory unit

FP adders

FP multipliers

Common data bus (CDB)

Observations

- The load buffers and store buffers hold data or addresses coming from and going to memory and behave like reservation stations
- The FP registers are connected by a pair of buses to the FUs and by a single bus to the store buffers
- All results form the FU and from memory are sent on the CDB (common data bus), which goes everywhere except to the load bufer
- All RSs have tag fields, employed by pipeline control

Key Idea

- Each FU has multiple reservation stations (RS)
- Issue to reservation stations was in-order (in-order issue)
- RS starts whenever they had collected source operands from real registers - hence out-of-order execution
- Reservation stations contain virtual registers (VR) that remove WAW and WAR induced stalls
  - RS fetches operands from RF and stores them into VR
  - Since virtual registers can be more than real registers, the technique can even eliminate hazards arising from name dependences that could not be eliminated by a compiler

Reservation Station Duties

- Each RS holds an instruction that has been issued and is awaiting execution at a FU, and either the operand values or the RS names that will provide the operand values
- RS fetches operands from CDB when they appear
- When all operands are present, enable the associated functional unit to execute
- Since values are not really written to registers
  - No WAW or WAR hazards are possible

Reservation Station Components

Op: Operation to perform in the unit (e.g., + or –)

Vj, Vk: Value of Source operands

- Store buffers has V field, result to be stored

Qj, Qk: Reservation stations producing source registers (value to be written)

- Note: Qj,Qk=0 => ready
- Store buffers only have Qi for RS producing result

•Note:
•max 1 valid Qj or Vj
•same with Qk or Vk

Busy: Indicates reservation station or FU is busy

A: information for memory address calculation for L/S

- Immediate → effective address

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Three Stages of Tomasulo Algorithm

1. Issue
   - Get the next instruction from the head of OP queue
     - The FIFO instruction queue (in-order issue)
   - If no RS is available
     - Structural hazards ➔ stall the pipeline
   - If there is an available RS
     - Issue the instruction
     - If the operands are available in the RFs
       - Fetch the operands and buffer them in the RS
       - To solve WAR hazards (register renaming)
     - If the operand is not available in the RFs
       - some FU is currently computing it
       - Redirect the operand source to that reservation station
       - To solve WAW hazards (register renaming)

Three Stages of Tomasulo Algorithm

2. Execute
   - If one of operands is not available
     - Monitor and wait for it
     - When the operand becomes available, it is placed into the corresponding RS
   - If all operands are available
     - The operation is performed at FU
     - RAW hazards are avoided !
     - Several insts. could become ready at the same clock cycle for the same FU

- Loads and stores require 2-step execution process
  - Effective address (EA) calculation, L/S buffer for memory access
  - L/S are maintained in program order through the EA calculation, which will help to prevent hazards through memory
- To preserve exception behavior
  - No instruction is allowed to initiate execution until all branches that precede it in program order have completed.

Three Stages of Tomasulo Algorithm

3. Write result
   - When result is available, write it on the CDB
   - When both the address and data values are available, they are sent to the memory unit

Summary for 3-stages of Tomasulo algorithm

1. Issue—get instruction from the head of Op Queue (FIFO)
   If reservation station free (no structural hazard), control issues instr & sends operands (renames registers).
2. Execute—operate on operands (EX)
   When both operands ready then execute; if not ready, watch Common Data Bus for result
3. Write result—finish execution (WB)
   Write on Common Data Bus to all awaiting units; mark reservation station available

- Normal data bus: data + destination ("go to" bus)
- Common data bus: data + source ("come from" bus)
  - 64 bits of data + 4 bits of Functional Unit source address
  - Write if matches expected Functional Unit (produces result)
  - Does the broadcast

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

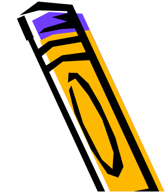
Example and the Algorithm

1. L.D F6, 34(R2)
2. L.D F2, 45(R3)
3. MUL.D F0, F2, F4
4. SUB.D F8, F2, F6
5. DIV.D F10, F0, F6
6. ADD.D F6, F8, F2

LD is 1 CC, ADDD/SUBD is 2 CC, MULT is 10 CC, and DIVD is 40 CC (Execution stage)

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Tomasulo Example



Instruction stream

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

3 Load/Buffers

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

FU count down

3 FP Adder R.S.
2 FP Mult R.S.

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
0	FU									

Clock cycle counter

Tomasulo Example Cycle 1

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
1	FU				Load1					

Tomasulo Example Cycle 2

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU		Load2		Load1					

Note: Can have multiple loads outstanding

Tomasulo Example Cycle 3

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	Mult1	Load2		Load1					

- Note: registers names are removed ("renamed") in Reservation Stations; MULT issued
- Load1 completing; what is waiting for Load1?

Tomasulo Example Cycle 4

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	Yes	SUBD	M(A1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
4	*FU*	Mult1	Load2		M(A1)	Add1				

- Load2 completing; what is waiting for Load2?

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Tomasulo Example Cycle 5

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
5	*FU*	Mult1	M(A2)		M(A1)	Add1	Mult2			

- Timer starts down for Add1, Mult1

Tomasulo Example Cycle 6

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	FU	Mult1	M(A2)		Add2	Add1	Mult2			

- Issue ADDD here despite name dependency on F6?

Tomasulo Example Cycle 7

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1	M(A2)		Add2	Add1	Mult2			

- Add1 (SUBD) completing; what is waiting for it?

Tomasulo Example Cycle 8

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
8	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 9

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
9	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 10

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
10	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

- Add2 (ADDD) completing; what is waiting for it?

Tomasulo Example Cycle 11

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
11	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- Write result of ADDD here?
- All quick instructions complete in this cycle!

Tomasulo Example Cycle 12

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
12	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 13

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
13	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 14

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
14	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 15

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
15	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- Mult1 (MULTD) completing; what is waiting for it?

Tomasulo Example Cycle 16

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

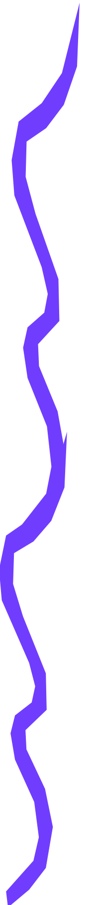
Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
16	FU	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- Just waiting for Mult2 (DIVD) to complete

Faster than light computation (skip a couple of cycles)



Tomasulo Example Cycle 55

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
1	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
55	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 56

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	FU	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- Mult2 (DIVD) is completing; what is waiting for it?

Tomasulo Example Cycle 57

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	57
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

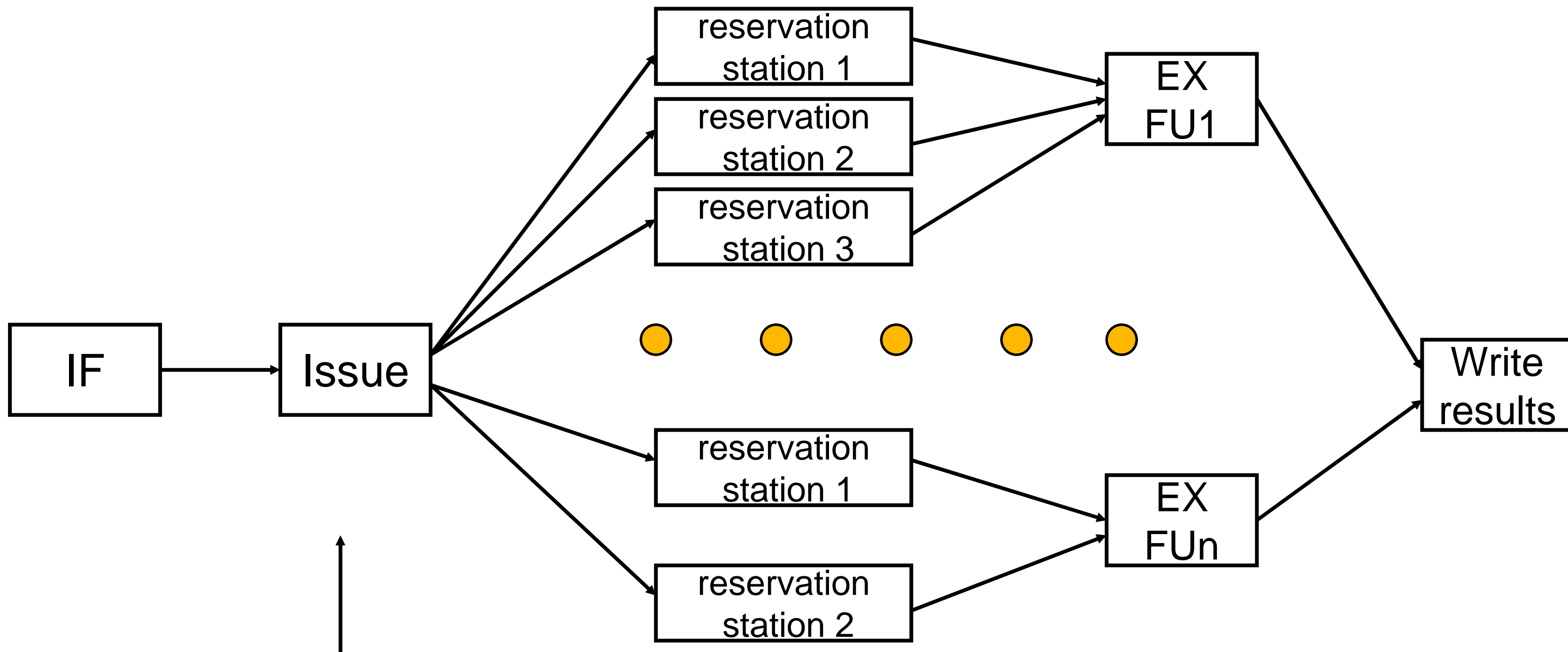
Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
57	FU	M*F4	M(A2)		(M-M+M	(M-M)	Result			

- Once again: In-order issue, out-of-order execution and out-of-order completion.

Tomasulo Solution



Structural hazard: delaying the issue until there is an empty reservation station

RAW data hazard: wait at the reservation station until the values of the source registers are available

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

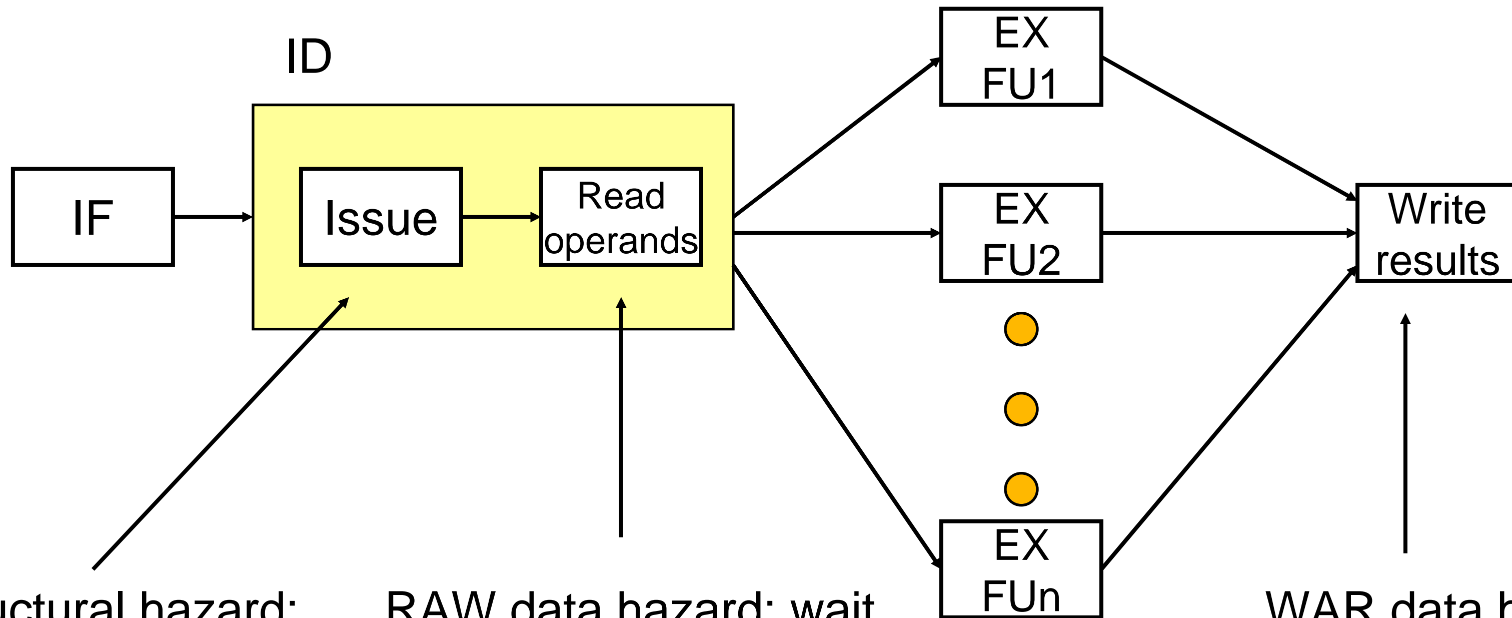
2 Major Advantages of Tomasulo

- Distribution of the hazard detection logic
  - Distributed RS and CDB
  - If multiple instructions are waiting on a single result, and each already has its other operand, then the instruction can be released simultaneously by the broadcast on CDB
  - If a centralized register file were used, the units would have to read their results from the registers when register buses are available
- Elimination of stalls for WAW and WAR
  - Rename register using RS
  - Store operands into RS as soon as they are available
  - For WAW-hazard, the last write will win

Tomasulo Drawbacks

- Complexity
  - delays of 360/91, MIPS 10000, Alpha 21264, IBM PPC 620 in CA:AQA 2/e, but not in silicon!
- Many associative stores (CDB) at high speed
- Performance limited by Common Data Bus
  - Each CDB must go to multiple functional units $\Rightarrow$high capacitance, high wiring density
  - Number of functional units that can complete per cycle limited to one!
    - Multiple CDBs $\Rightarrow$ more FU logic for parallel assoc stores
- Non-precise interrupts!
  - We will address this later

Scoreboard Solution



Structural hazard: delaying the issue
WAW data hazard: delaying the issue

RAW data hazard: wait until the values of the source registers are available in the registers

WAR data hazard: delaying the write if a WAR hazard exists

Tamasulo vs. Scoreboard

- Instruction stall
  - Tomasulo stalls only for structural hazards
  - Scoreboard stalls for both structural and WAW hazards
- Operand fetch
  - Tomasulo uses RS as operand buffers
  - Scoreboard uses centralized RF
    - No forwarding and WAR hazards are possible
- Write back
  - Scoreboard has to stall writes to resolve WAR hazards
  - Tamasulo uses renaming register to resolve WAR and WAW hazards

The Load and Store Problem

Dynamic memory disambiguation

- A load and store can safely be done in different order, provided that they access different addresses
- If load and store accesses the same address
  - WAR hazard, the load is before the store in program order
  - RAW hazard, the store is before the load in program order
  - WAW hazard, the store is before the other store
- To detect such hazards, data memory address (EA) must be calculated in program order
- We really only need to keep the relative order between stores and other memory references (Load can be reordered freely)
  - Conflicting stores cannot be reordered with respect to either a load or a store

Tomasulo Loop Example

Loop:	L.D	F0	0	R1
	MUL.D	F4	F0	F2
	S.D	F4	0	R1
	SUBI	R1	R1	#8
	BNEZ	R1	Loop	

- Assume Multiply takes 4 clocks
- Assume first load takes 8 clocks (cache miss?), second load takes 4 clocks (hit)
- To be clear, will show clocks for SUBI, BNEZ
- Reality, integer instructions ahead

Loop Example Cycle 0

Instruction status						Execution	Write			
Instruction		j	k	iteration	Issue	complete	Result		Busy	Address
LD	F0	0	R1	1				Load1	No	
MULT	F4	F0	F2	1				Load2	No	
SD	F4	0	R1	1				Load3	No	Qi
LD	F0	0	R1	2				Store1	No	
MULT	F4	F0	F2	2				Store2	No	
SD	F4	0	R1	2				Store3	No	

Reservation Stations				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	No					
0	Mult2	No					

Code:

LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
0	**80**	Qi								

Loop Example Cycle 1

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result		Busy	Address
LD	F0	0	R1	1	1			Load1	Yes	80
MULT	F4	F0	F2	1				Load2	No	
SD	F4	0	R1	1				Load3	No	
LD	F0	0	R1	2						
MULT	F4	F0	F2	2						
SD	F4	0	R1	2						

	Busy	Qi
Store1	No	
Store2	No	
Store3	No	

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	No					
0	Mult2	No					

Code:

```
LD    F0    0   R1
MULT  F4    F0  F2
SD    F4    0   R1
SUBI  R1    R1  #8
BNEZ  R1    Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
1	80	Qi	Load1								

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Loop Example Cycle 2

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1		
MULT F4	F0	F2	1	2		
SD F4	0	R1	1			
LD F0	0	R1	2			
MULT F4	F0	F2	2			
SD F4	0	R1	2			

	Busy	Address	Qi
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	No					

Code:

```
LD    F0   0   R1
MULT  F4   F0  F2
SD    F4   0   R1
SUBI  R1   R1  #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
2	80	Qi	Load1		Mult1					

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Loop Example Cycle 3

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1		
MULT F4	F0	F2	1	2		
SD F4	0	R1	1	3		
LD F0	0	R1	2			
MULT F4	F0	F2	2			
SD F4	0	R1	2			

	Busy	Address	Qi
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	No					

Code:

```
LD   F0  0  R1
MULT F4  F0 F2
SD   F4  0  R1
SUBI R1  R1 #8
BNEZ R1  Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
3	80	Qi	Load1		Mult1					

- Note: MULT1 has no registers names in RS

Loop Example Cycle 4

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1		
MULT F4	F0	F2	1	2		
SD F4	0	R1	1	3		
LD F0	0	R1	2			
MULT F4	F0	F2	2			
SD F4	0	R1	2			

	Busy	Address	Qi
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	No					

Code:

```
LD    F0   0  R1
MULT  F4   F0 F2
SD    F4   0  R1
SUBI  R1   R1 #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
4	72	Qi	Load1		Mult1					

Loop Example Cycle 5

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result		Busy	Address
LD F0	0	R1	1	1			Load1	Yes	80
MULT F4	F0	F2	1	2			Load2	No	
SD F4	0	R1	1	3			Load3	No	
LD F0	0	R1	2						
MULT F4	F0	F2	2						
SD F4	0	R1	2						

	Busy	Address	Qi
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	No					

Code:

```
LD    F0   0  R1
MULT  F4   F0 F2
SD    F4   0  R1
SUBI  R1   R1 #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
5	72	Qi	Load1		Mult1					

Loop Example Cycle 6

Instruction status						Execution	Write					
Instruction		j	k	iteration	Issue	complete	Result		Busy	Address		
LD	F0	0	R1	1	1			Load1	Yes	80		
MULT	F4	F0	F2	1	2			Load2	Yes	72		
SD	F4	0	R1	1	3			Load3	No		Qi	
LD	F0	0	R1	2	6			Store1	Yes	80	Mult1	
MULT	F4	F0	F2	2				Store2	No			
SD	F4	0	R1	2				Store3	No			
Reservation Stations					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	0	Mult1	Yes	MULTD		R(F2)	Load1		SUBI	R1	R1	#8
	0	Mult2	No						BNEZ	R1	Loop	
Register result status												
Clock		R1		F0	F2	F4	F6	F8	F10	F12	...	F30
6		72	Qi	Load2		Mult1						

- Note: F0 never sees Load1 result

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Loop Example Cycle 7

Instruction status						Execution	Write					
Instruction		j	k	iteration	Issue	complete	Result		Busy	Address		
LD	F0	0	R1	1	1			Load1	Yes	80		
MULT	F4	F0	F2	1	2			Load2	Yes	72		
SD	F4	0	R1	1	3			Load3	No		Qi	
LD	F0	0	R1	2	6			Store1	Yes	80	Mult1	
MULT	F4	F0	F2	2	7			Store2	No			
SD	F4	0	R1	2				Store3	No			
Reservation Stations					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	0	Mult1	Yes	MULTD		R(F2)	Load1		SUBI	R1	R1	#8
	0	Mult2	Yes	MULTD		R(F2)	Load2		BNEZ	R1	Loop	
Register result status												
Clock		R1		F0	F2	F4	F6	F8	F10	F12 ...		F30
7		72	Qi	Load2		Mult2						

- Note: MULT2 has no registers names in RS

Loop Example Cycle 8

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1		
MULT F4	F0	F2	1	2		
SD F4	0	R1	1	3		
LD F0	0	R1	2	6		
MULT F4	F0	F2	2	7		
SD F4	0	R1	2	8		

	Busy	Address	Qi
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	Yes	MULTD		R(F2)	Load2	

Code:

```
LD    F0   0  R1
MULT  F4   F0 F2
SD    F4   0  R1
SUBI  R1   R1 #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
8	72	Qi	Load2		Mult2					

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Loop Example Cycle 9

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result		Busy	Address	
LD F0	0	R1	1	1	9		Load1	Yes	80	
MULT F4	F0	F2	1	2			Load2	Yes	72	
SD F4	0	R1	1	3			Load3	No		Qi
LD F0	0	R1	2	6			Store1	Yes	80	Mult1
MULT F4	F0	F2	2	7			Store2	Yes	72	Mult2
SD F4	0	R1	2	8			Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	Yes	MULTD		R(F2)	Load2	

Code:

```
LD   F0  0  R1
MULT F4  F0 F2
SD   F4  0  R1
SUBI R1  R1 #8
BNEZ R1  Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
9	64	Qi	Load2		Mult2					

- Load1 completing; what is waiting for it?

Loop Example Cycle 10

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1	9	10
MULT F4	F0	F2	1	2		
SD F4	0	R1	1	3		
LD F0	0	R1	2	6	10	
MULT F4	F0	F2	2	7		
SD F4	0	R1	2	8		

	Busy	Address	Qi
Load1	No		
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS for j Qj	RS for k Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
4	Mult1	Yes	MULTD	M(80)	R(F2)		
0	Mult2	Yes	MULTD		R(F2)	Load2	

Code:

```
LD    F0    0  R1
MULT  F4    F0 F2
SD    F4    0  R1
SUBI  R1    R1 #8
BNEZ  R1    Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
10	**64**	Qi	Load2		Mult2					

- Load2 completing; what is waiting for it?

Loop Example Cycle 11

Instruction status						Execution	Write					
Instruction		j	k	iteration	Issue	complete	Result		Busy	Address		
LD	F0	0	R1	1	1	9	10	Load1	No			
MULT	F4	F0	F2	1	2			Load2	No			
SD	F4	0	R1	1	3			Load3	Yes	64	Qi	
LD	F0	0	R1	2	6	10	11	Store1	Yes	80	Mult1	
MULT	F4	F0	F2	2	7			Store2	Yes	72	Mult2	
SD	F4	0	R1	2	8			Store3	No			
Reservation Stations					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	3	Mult1	Yes	MULTD	M(80)	R(F2)			SUBI	R1	R1	#8
	4	Mult2	Yes	MULTD	M(72)	R(F2)			BNEZ	R1	Loop	
Register result status												
Clock		R1		F0	F2	F4	F6	F8	F10	F12	...	F30
11		64	Qi	Load3		Mult2						

Loop Example Cycle 12

Instruction status						Execution	Write					
Instruction		j	k	iteration	Issue	complete	Result		Busy	Address		
LD	F0	0	R1	1	1	9	10	Load1	No			
MULT	F4	F0	F2	1	2			Load2	No			
SD	F4	0	R1	1	3			Load3	Yes	64	Qi	
LD	F0	0	R1	2	6	10	11	Store1	Yes	80	Mult1	
MULT	F4	F0	F2	2	7			Store2	Yes	72	Mult2	
SD	F4	0	R1	2	8			Store3	No			
Reservation Stations					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	2	Mult1	Yes	MULTD	M(80)	R(F2)			SUBI	R1	R1	#8
	3	Mult2	Yes	MULTD	M(72)	R(F2)			BNEZ	R1	Loop	
Register result status												
Clock		R1		F0	F2	F4	F6	F8	F10	F12	...	F30
12		64	Qi	Load3		Mult2						

Loop Example Cycle 13

Instruction status						Execution	Write					
Instruction		j	k	iteration	Issue	complete	Result		Busy	Address		
LD	F0	0	R1	1	1	9	10	Load1	No			
MULT	F4	F0	F2	1	2			Load2	No			
SD	F4	0	R1	1	3			Load3	Yes	64	Qi	
LD	F0	0	R1	2	6	10	11	Store1	Yes	80	Mult1	
MULT	F4	F0	F2	2	7			Store2	Yes	72	Mult2	
SD	F4	0	R1	2	8			Store3	No			
Reservation Stations					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	1	Mult1	Yes	MULTD	M(80)	R(F2)			SUBI	R1	R1	#8
	2	Mult2	Yes	MULTD	M(72)	R(F2)			BNEZ	R1	Loop	
Register result status												
Clock		R1		F0	F2	F4	F6	F8	F10	F12 ...		F30
13		64	Qi	Load3		Mult2						

Loop Example Cycle 14

Instruction status						*Execution*	*Write*					
Instruction		*j*	*k*	*iteration*	*Issue*	*complete*	*Result*		Busy	Address		
LD	F0	0	R1	1	1	9	10	Load1	No			
MULT	F4	F0	F2	1	2	14		Load2	No			
SD	F4	0	R1	1	3			Load3	Yes	64	Qi	
LD	F0	0	R1	2	6	10	11	Store1	Yes	80	Mult1	
MULT	F4	F0	F2	2	7			Store2	Yes	72	Mult2	
SD	F4	0	R1	2	8			Store3	No			
Reservation Stations					*S1*	*S2*	*RS for j*	*RS for k*				
	Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*	*Code:*			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	0	Mult1	Yes	MULTD	M(80)	R(F2)			SUBI	R1	R1	#8
	1	Mult2	Yes	MULTD	M(72)	R(F2)			BNEZ	R1	Loop	
Register result status												
Clock		**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12 ...*		*F30*
14		**64**	*Qi*	Load3		Mult2						

- Mult1 completing; what is waiting for it?

Loop Example Cycle 15

Instruction status						Execution	Write					
Instruction		j	k	iteration	Issue	complete	Result		Busy	Address		
LD	F0	0	R1	1	1	9	10	Load1	No			
MULT	F4	F0	F2	1	2	14	15	Load2	No			
SD	F4	0	R1	1	3			Load3	Yes	64	Qi	
LD	F0	0	R1	2	6	10	11	Store1	Yes	80	M(80)*R(F	
MULT	F4	F0	F2	2	7	15		Store2	Yes	72	Mult2	
SD	F4	0	R1	2	8			Store3	No			
Reservation Stations					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	0	Mult1	No						SUBI	R1	R1	#8
	0	Mult2	Yes	MULTD	M(72)	R(F2)			BNEZ	R1	Loop	
Register result status												
Clock		R1		F0	F2	F4	F6	F8	F10	F12	...	F30
15		64	Qi	Load3		Mult2						

- Mult2 completing; what is waiting for it?

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Loop Example Cycle 16

Instruction status						Execution	Write					
Instruction		j	k	iteration	Issue	complete	Result		Busy	Address		
LD	F0	0	R1	1	1	9	10	Load1	No			
MULT	F4	F0	F2	1	2	14	15	Load2	No			
SD	F4	0	R1	1	3			Load3	Yes	64	Qi	
LD	F0	0	R1	2	6	10	11	Store1	Yes	80	M(80)*R(F	
MULT	F4	F0	F2	2	7	15	16	Store2	Yes	72	M(72)*R(7	
SD	F4	0	R1	2	8			Store3	No			
Reservation Stations					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	0	Mult1	Yes	MULTD		R(F2)	Load3		SUBI	R1	R1	#8
	0	Mult2	No						BNEZ	R1	Loop	
Register result status												
Clock		R1		F0	F2	F4	F6	F8	F10	F12	...	F30
16		64	Qi	Load3		Mult1						

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Loop Example Cycle 17

Instruction status						Execution	Write					
Instruction		*j*	*k*	*iteration*	*Issue*	*complete*	*Result*		Busy	Address		
LD	F0	0	R1	1	1	9	10	Load1	No			
MULT	F4	F0	F2	1	2	14	15	Load2	No			
SD	F4	0	R1	1	3			Load3	Yes	64	Qi	
LD	F0	0	R1	2	6	10	11	Store1	Yes	80	M(80)*R(F	
MULT	F4	F0	F2	2	7	15	16	Store2	Yes	72	M(72)*R(7	
SD	F4	0	R1	2	8			Store3	Yes	64	Mult1	
Reservation Stations					*S1*	*S2*	*RS for j*	*RS for k*				
	Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*	*Code:*			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	0	Mult1	Yes	MULTD		R(F2)	Load3		SUBI	R1	R1	#8
	0	Mult2	No						BNEZ	R1	Loop	
Register result status												
Clock		**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
17		**64**	*Qi*	Load3		Mult1						

Loop Example Cycle 18

Instruction status						Execution	Write					
Instruction		j	k	iteration	Issue	complete	Result		Busy	Address		
LD	F0	0	R1	1	1	9	10	Load1	No			
MULT	F4	F0	F2	1	2	14	15	Load2	No			
SD	F4	0	R1	1	3	18		Load3	Yes	64	Qi	
LD	F0	0	R1	2	6	10	11	Store1	Yes	80	M(80)*R(F	
MULT	F4	F0	F2	2	7	15	16	Store2	Yes	72	M(72)*R(7	
SD	F4	0	R1	2	8			Store3	Yes	64	Mult1	
Reservation Stations					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	0	Mult1	Yes	MULTD		R(F2)	Load3		SUBI	R1	R1	#8
	0	Mult2	No						BNEZ	R1	Loop	
Register result status												
Clock		**R1**		F0	F2	F4	F6	F8	F10	F12	...	F30
18		**56**	Qi	Load3		Mult1						

Loop Example Cycle 19

Instruction status						Execution	Write					
Instruction		j	k	iteration	Issue	complete	Result		Busy	Address		
LD	F0	0	R1	1	1	9	10	Load1	No			
MULT	F4	F0	F2	1	2	14	15	Load2	No			
SD	F4	0	R1	1	3	18	19	Load3	Yes	64	Qi	
LD	F0	0	R1	2	6	10	11	Store1	No			
MULT	F4	F0	F2	2	7	15	16	Store2	Yes	72	M(72)*R(7	
SD	F4	0	R1	2	8			Store3	Yes	64	Mult1	
Reservation Stations					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	0	Mult1	Yes	MULTD		R(F2)	Load3		SUBI	R1	R1	#8
	0	Mult2	No						BNEZ	R1	Loop	
Register result status												
Clock		R1		F0	F2	F4	F6	F8	F10	F12	...	F30
19		56	Qi	Load3		Mult1						

Loop Example Cycle 20

Instruction status						Execution	Write					
Instruction		j	k	iteration	Issue	complete	Result		Busy	Address		
LD	F0	0	R1	1	1	9	10	Load1	No			
MULT	F4	F0	F2	1	2	14	15	Load2	No			
SD	F4	0	R1	1	3	18	19	Load3	Yes	64	Qi	
LD	F0	0	R1	2	6	10	11	Store1	No			
MULT	F4	F0	F2	2	7	15	16	Store2	Yes	72	M(72)*R(7	
SD	F4	0	R1	2	8	20		Store3	Yes	64	Mult1	
Reservation Stations					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	0	Mult1	Yes	MULTD		R(F2)	Load3		SUBI	R1	R1	#8
	0	Mult2	No						BNEZ	R1	Loop	
Register result status												
Clock		R1		F0	F2	F4	F6	F8	F10	F12	...	F30
20		56	Qi	Load3		Mult1						

Loop Example Cycle 21

Instruction status						Execution	Write					
Instruction		*j*	*k*	*iteration*	*Issue*	*complete*	*Result*		Busy	Address		
LD	F0	0	R1	1	1	9	10	Load1	No			
MULT	F4	F0	F2	1	2	14	15	Load2	No			
SD	F4	0	R1	1	3	18	19	Load3	Yes	64	Qi	
LD	F0	0	R1	2	6	10	11	Store1	No			
MULT	F4	F0	F2	2	7	15	16	Store2	No			
SD	F4	0	R1	2	8	20	21	Store3	Yes	64	Mult1	
Reservation Stations					*S1*	*S2*	*RS for j*	*RS for k*				
	Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*	*Code:*			
	0	Add1	No						LD	F0	0	R1
	0	Add2	No						MULT	F4	F0	F2
	0	Add3	No						SD	F4	0	R1
	0	Mult1	Yes	MULTD		R(F2)	Load3		SUBI	R1	R1	#8
	0	Mult2	No						BNEZ	R1	Loop	
Register result status												
Clock		**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
21		**56**	*Qi*	Load3		Mult1						

Tomasulo Summary

- Reservations stations: renaming to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards of Scoreboard
  - Allows loop unrolling in HW
- For one CDB, only one operation can use it at a single clock cycle
- Not limited to basic blocks (integer units gets ahead, beyond branches)
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- 360/91 descendants are Pentium II; PowerPC 604; MIPS R10000; HP-PA 8000; Alpha 21264