Chapter 5a: Instruction Level Parallelism

Static Scheduling

Dynamic Scheduling

- Scoreboarding
- Tomasulo's Algorithm
- Interrupts Again

Dynamic Branch Prediction

Static Scheduling

Basic idea

- Have compiler reorder code to mitigate the effect of data and control dependencies

Three Examples

- Loop unrolling
- Software pipelining
- Trace scheduling

Loop Unrolling

With scheduling

```
Loop:  LD    F0, 0(R1)
       stall
       ADDD  F4, F0, F2
       SUB   R1, R1, #8
       BNEZD R1, Loop     ; Assume delayed branch
       SD    8(R1), F4
```


Loop Unrolling

Unrolling the loop

```
Loop:  LD    F0, 0(R1)
       ADDD  F4, F0, F2
       SD    0(R1), F4
       LD    F6, -8(R1)
       ADDD  F8, F6, F2
       SD    -8(R1), F8
       LD    F10, -16(R1)
       ADDD  F12, F10, F2
       SD    -16(R1), F12
       LD    F14, -24(R1)
       ADDD  F16, F14, F2
       SD    -24(R1), F16
       SUB   R1, R1, #32
       BNEZ R1, Loop     ; Assume delayed branch
```

Loop Unrolling

Unrolling the loop

```
Loop:  LD    F0, 0(R1)
       LD    F6, -8(R1)
       LD    F10, -16(R1)
       LD    F14, -24(R1)
       ADDD  F4, F0, F2
       ADDD  F8, F6, F2
       ADDD  F12, F10, F2
       ADDD  F16, F14, F2
       SD    0(R1), F4
       SD    -8(R1), F8
       SD    -16(R1), F12
       SD    -24(R1), F16
       SUB   R1, R1, #32
       BNEZ R1, Loop     ; Assume delayed branch
```


Software Pipelining [Charlesworth 1981]

```
sum = 0.0;
for (i=1; i<=N; i++) {
       ; sum = sum + a[i]*b[i]
       load a[i]          ; Ai
       load b[i]          ; Bi
       mult ab[i]         ; *i
       add  sum[i]        ; +i
}
```

```
sum = 0.0;
START-UP-BLOCK
for (i=3; i<=N; i++) {
       load a[i]          ; Ai
       load b[i]          ; Bi
       mult ab[i-1]       ; *i-1
       add  sum[i-2]      ; +i-2
}
FINISH-UP-BLOCK
```

```
                  LOOP
START-UP    i=3   ...  i=N   FINISH-UP
--------    ---        ---   ---------
A1    A2    A3    Ai    AN
B1    B2    B3    Bi    BN
      *1    *2    *i-1  *N-1  *N
            +1    +i-2  *N-2  +N-1  +N
```


Trace Scheduling [Ellis 1985]

```
b[i] = ‘‘old’’
a[i] =
if (a[i] > 0) then
       b[i] = ‘‘new’’;  common case
else
       X
endif
c[i] =
```

Until done

- Select most common path -- a trace
- Schedule trace across basic blocks
- Repair other paths

```
trace to be              repair
scheduled:               code:
b[i] = ‘‘old’’           A:  restore old b[i]
a[i] =                       X
b[i] = ‘‘new’’               maybe recalculate c[i]
c[i] =                       goto B
if (a[i] <= 0) goto A
B:
```


Static Scheduling Summary

Loop unrolling

- \+ larger block for scheduling
- \+ reduces branch frequency
- \- expands code size
- \- have to handle "extra" iterations

Software pipelining

- \+ no dependences in loop body
- \- does not reduce branch frequency
- \- need start-up & finish up blocks

Trace scheduling

- \+ works for loops
- \- more complex than unrolling & pipelining
- \- does not seem to handle more-general cases after all

Dynamic Scheduling

Run faster while preserving illusion of sequential execution

Data dependencies: RAW
(in registers, memory, condition codes, etc.)

Name dependencies: WAR & WAW

Trap, interrupts, & exceptions

Control dependence: branches & jumps

Dynamic Scheduling, cont.

The situation:

```
DIVF    F0, F2, F4
ADDF    F10, F0, F8
MULTF   F8, F8, F14
```

The Problem:

ADDF stalls due to RAW hazard

MULTF stalls because ADDF stalls

Example

```
      1   2   3   4   5   6   7   8   9  10  11
DIVF             IF  ID  E/  E/  E/  E/  MEM WB
ADDF                 IF  ID  **  **  **  E+  E+
MULTF    why stall? -->  IF  **  **  **  ID  E*
```

In-Order execution limits performance

Dynamic Scheduling, cont.

Solutions

- Static Scheduling
- Dynamic Scheduling

Static Scheduling (Software)

- Compiler reorganizes instructions
- Burden on compiler
- Simpler hardware
- Can use more powerful algorithms

Dynamic Scheduling (Hardware)

- Hardware reorganizes instructions
- Handles dependences unknown at compile time
- Software is more portable
- More complex hardware

Dynamic Scheduling, cont.

In-Order Issue

- Instructions issued sequentially.
- Stall instruction $i+1$ if instruction i stalls

Out-of-order Issue (Execution)

- Issue independent instructions as soon as possible

Split issue into two parts

- Check for structural hazards
- Wait for data dependences

Dynamic Scheduling, cont.

Original DLX pipeline

ID -- decode, check all hazards, read operands

EX -- execute

Dynamic DLX pipeline

Issue -- decode, check structural hazards

ReadOps -- wait until data hazards clear
then read operands

Execute -- execute

Dynamic Scheduling, cont.

WAR Hazards

May Exist with dynamic scheduling

```
DIVF    F0, F2, F4
ADDF    F10, F0, F8
MULTF   F8, F8, F14
```

A rule to avoid WAR hazards

Read registers during ReadOps

Queue both ADDF operation and copies of operands

For example, within ReadOps

if src1 ready and tmp1 empty, tmp1 ← src1

if src2 ready and tmp2 empty, tmp2 ← src2

if both tmp1 and tmp2 full,

issue dest ← tmp1 op tmp2

Dynamic Scheduling, cont.

WAW Hazards

May Exist with dynamic scheduling

```
DIVF    F0, F2, F4
ADDF    F10, F0, F8
MULTF   F10, F8, F14
```

A rule to avoid WAW hazards

Stall all instruction issue

More aggressive solution later.

Scoreboard

Centralized control scheme

Controls all instruction issue

Detects all hazards

Implemented in CDC 6600 in 1964

CDC 6600 had 18 separate function units

4 FP units: 2 multiply, 1 add, 1 divide

7 memory units: 5 load, 2 store

7 integer units: add, shift, logical, etc.

Dynamic DLX

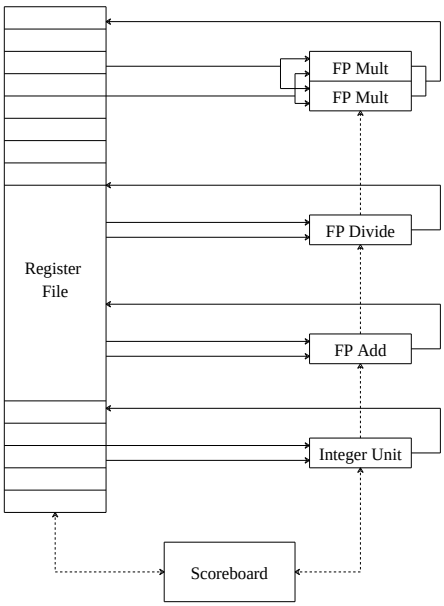
2 FP multiply

1 FP adder

1 FP divide

1 integer unit

Dynamic DLX Datapath



Dynamic DLX Pipeline

Ignore IF and MEM for simplicity

4-stage execution

Issue	Check for structural hazards Check for WAW hazards Stall all issue until clear
ReadOps	Check for RAW hazards Wait until operands are ready Read Registers
Execute	Execute operation Notify scoreboard when complete
Write	Check for WAR hazards Stall write until clear

A completing instruction cannot write *dest* if an earlier instruction has not read *dest*

Scoreboard

Three data structures

Instruction Status
Function Unit Status
Register Result Status

Instruction Status

Indicates which execution stage each instruction is currently in

Function Unit Status

Busy	Indicate whether function unit is busy
Op	Operation (e.g., add or subtract)
F_i	Destination register
F_j, F_k	Source registers containing src1 and src2
Q_j, Q_k	Function units producing src1 and src2
R_j, R_k	Ready bits for F_j, F_k

Register Result Status

Indicates which function unit (if any) will write a register.

Scoreboard Example

Example code

```
LF      F6,34(R2)
LF      F2,45(R3)
MULTF   F0,F2,F4
SUBF    F8,F6,F2
DIVF    F10,F0,F6
ADDF    F6,F8,F2
```

Scoreboard Example

Instruction Status					
Instruction		Issue	ReadOps	Execute	Write
LF	F6,34(R2)				
LF	F2,45(R3)				
MULTF	F0,F2,F4				
SUBF	F8,F6,F2				
DIVF	F10,F0,F6				
ADDF	F6,F8,F2				

Functional Unit Status									
FU	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
1	Integer									
2	Mult1									
3	Mult2									
4	Add									
5	Divide									

Register Result Status								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU									

Scoreboard Example

Speedup

- 1.7 for FORTRAN programs
- 2.5 for hand-coded assembly language
  - Effect of modern compilers????

Hardware

- Scoreboard was approximately one FU
- Main cost was busses (4X)

Limitations

- No bypassing
- Data dependences are handled through registers
- WAW hazards cause stalls
- Limits ability to dynamically unroll loops

Dynamic Scheduling, cont.

Consider the following example:

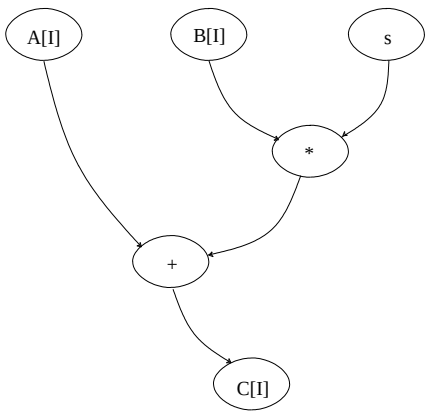
```
FORTRAN:
      DO I = 1, N
           C[I] = A[I] + s * B[I]


ASSEMBLY:
      LF      F0, A(R1)
      LF      F2, B(R1)
      MULTF   F2, F2, F4          /* s in F4 */
      ADDF    F2, F2, F0
      SF      C(R1), F2
```

What would Scoreboarding do?

- How could we do better?

Dynamic Scheduling, cont.

Data flow graph:



Registers are *Names* for data values.

- Think of registers specifiers as *tags*
- NOT storage locations.

IBM 360/91

Fast 360 for scientific code

- Completed in 1967
- Predates cache memories

Pipelined, rather than multiple, function units

- Adder supports 3 instructions
- Multiplier supports 2 instructions

Differences from Scoreboard

- Distributed hazard detection and control
- Results are bypassed to function units
- Common Data Bus (CDB) for results

Dynamic DLX-T

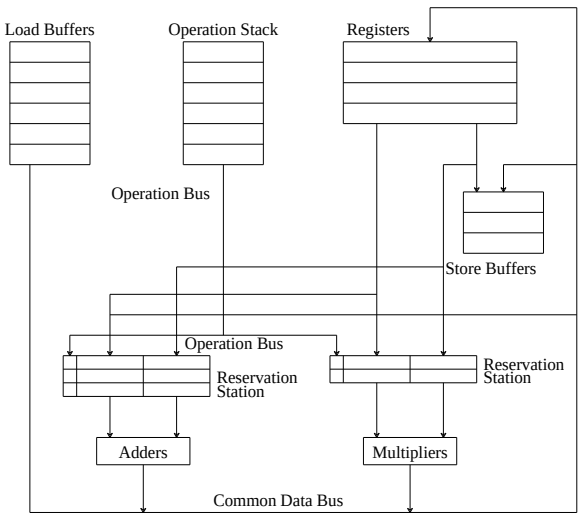
Tomasulo's algorithm

- Register renaming
- Uses ''tags'' to pass data values
- *Reservation stations* distribute control
- CDB broadcasts all results.

Extend DLX as example

- Assume multiple FUs rather than pipelined
- Main difference is Register-Memory Instructions

Dynamic DLX-T



Dynamic DLX-T Issue

3-stage Execution (ignore IF and MEM)

Issue	Get instruction from queue ALU Op: Check for available reservation stati Load: Check for available load buffer If not stall, due to structural hazard
Execute	If operands available, execute operation If not, monitor CDB for operand
Write	If CDB available, write it on CDB If not, stall.

Similar to Scoreboard, BUT

- No checking for WAW or WAR
- CDB broadcasts results to function units, not just registers

Dynamic DLX-T, cont

Reservation Stations

Handle distributed hazard detection and instruction control

Everything, except load buffers, have a *tag*

4-bit tag specifies reservation station or load buffer

Specifies which FU will produce result

Register specifier is used to assign tags

THEN IT'S DISCARDED!

Register specifiers are ONLY used in ISSUE

Dynamic DLX-T, cont

Reservation Stations

Op	Opcode
Q_j,Q_k	Tag fields
V_j,V_k	Operand values
Busy	Currently in use

Register File and Store Buffer

Q_i	Tag field
Busy	Currently in use

Load Buffers

Busy	Currently in use

Tomasulo Example

Example code

```
LF      F6,34(R2)
LF      F2,45(R3)
MULTF   F0,F2,F4
SUBF    F8,F6,F2
DIVF    F10,F0,F6
ADDF    F6,F8,F2
```


Tomasulo Example

Instruction Status (For illustration ONLY)				
Instruction		Issue	Execute	Write
LF	F6,34(R2)			
LF	F2,45(R3)			
MULTF	F0,F2,F4			
SUBF	F8,F6,F2			
DIVF	F10,F0,F6			
ADDF	F6,F8,F2			

Reservation Stations							
FU	Name	Busy	Op	Vj	Vk	Qj	Qk
1	Add1						
2	Add2						
3	Add3						
4	Mult1						
5	Mult2						

Register Result Status									
	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi									
Busy									

Tomasulo, cont

Out-of-order loads and stores?

- What about WAW, RAW, and WAR hazards?
- Compare all load addresses against the address in the store buffers.
- Stall if they match

CDB is a bottleneck

- Could duplicate
- Increases the required hardware

Complex implementation

Tomasulo, cont

Advantages

- Distribution of hazard detection
- Elimination of WAR and WAW stalls

Common Data Bus

- + Broadcasts results to multiple instructions
- - Central bottleneck

Register Renaming

- Eliminates WAR and WAW Hazards
- Allows dynamic loop unrolling
  - Especially important with only 4 registers
- Requires many associative lookups

A Comparison

Feature	**Scoreboarding**	**Tomasulo's**
Structural	Stall in Issue for FU	Stall in Issue for RS
RAW	Thru registers	From CDB
WAR	Stall in Write	Copy operand to RS
WAW	Stall in Issue	Register Renaming
Logic	Centralized	Decentralized
Bottlenecks	No register bypass; stalls in Issue block pipe	One result bus

Problems with Dynamic Scheduling

Consider our canonical code fragment:

```
LF      F6,34(R2)
LF      F2,45(R3)
MULTF   F0,F2,F4
SUBF    F8,F6,F2
DIVF    F10,F0,F6
ADDF    F6,F8,F2
```

What happens if DIVF causes an interrupt?

- ADDF has already completed

What happens if the first LF gets a page fault?

- MULTF and ADDF may have completed

Out-of-order completion makes interrupts hard

Issue/Execution/Completion

Issue

- Send instruction to execution unit or reservation station

Execution

- *Begin* execution

Completion

- Change architectural state
- Make results visible

In-Order vs. Out-of-Order

- In-Order means sequential order
- Out-of-Order means not sequential

Examples

Vanilla DLX

- In-order everything

Multi-cycle DLX

- In-order Issue
- In-order Exception
- Out-of-order Completion

Dynamic DLX (Scoreboard)

- In-order Issue
- Out-of-order Execution
- Out-of-order Completion

Dynamic DLX-T (Tomasulo)??

Solutions

Imprecise Interrupts

- Ignore the problem
- IEEE Standard disallows it

In-order completion

- Stall pipeline when necessary

Software clean-up

- Save information for trap handlers
- Machine dependent

Hardware clean-up

- Restore to consistent state

Re-order instructions

- Complete Out-of-Order
- *Retire* In-Order

Model Architecture

Result Shift Register (RSR)

- Control result bus
- At most one completion per cycle

Stage	FU	Dest	Valid	PC
1				
2	INT ADD	0	1	7
3				
4				
5	FP ADD	4	1	6
N				

An instruction taking ***i*** cycles reserves stage ***i*** when it issues

- Stalls if stage ***i*** is already taken

Shift down each cycle

Instruction is stage 1 gets result bus

In-Order Completion

Reserve Preceding stages

An instruction taking ***j*** stages reserves all available stages ***i*** < ***j***

Prevents a later (sequential) instruction from completing before it.

Must synchronize stores

(1) Wait for RSR to empty

(2) Put dummy store in RSR

What if store causes exception??

Re-order Buffer

Overview

Instructions complete out-of-order

Re-order buffer reorganizes instructions

Modify state in-order

Re-order Buffer (RB)

	Entry	Dest	Result	Excep	Valid	PC
	1					
head ->	2	4			0	6
	3	0			0	7
tail ->	4					
	5					
	N					

Add tag to RSR to point to corresponding RB entry.

Re-order Buffer

On Issue, allocate RB entry and RSR entry

Make RSR point to RB

On completion, put result in RB entry pointed to by RSR tag.

Mark RB entry valid

When head of RB is valid, update register.

Hardware

Need bypass paths from RB

Multiple bypass paths???

History Buffer

Overview

Instructions complete out-of-order

Register file updated immediately

Hardware cleans-up on exception

History Buffer (HB)

Logs previous value of register

Used to *undo* instructions

	Entry	Dest	Value	Excep	Valid	PC
	1					
head ->	2	4	408000		0	6
	3	0	42		0	7
tail ->	4					
	5					
	N					

History Buffer

On Issue, copy current value of dst reg to HB.

On completion, mark HB entry valid.

On exception, mark exception bit.

When head of HB contains an entry marked valid, reuse entry (increment head).

When entry has exception

- Stall issue
- Wait for pipeline to empty
- Rollback register file from tail to head
- PC at head is precise PC

Hardware

- 3 read ports
- Bypassing is more slightly more complex

Future File

Overview

- Separate register files
- Architectural file updated in-order
- Future file updated out-of-order
- Re-order buffer controls updates to architectural file

Future File

Future file is managed like an ordinary imprecise pipeline

Architecture file managed like re-order buffer scheme

- When head entry valid, update architectural file

On exception,

- Architectural file contains precise state

Advantages

- State saving is easy
- No extra bypassing

Performance Comparison

Simulations using the Livermore Loops

Compare three options

- In-order completion
- Reorder buffer
- Reorder buffer with bypass
  - Same as Future file and History file

Examine both ways of handling stores

- Blocking
- Non-blocking

Relative performance

$$\frac{\text{Time w/ precise interrupts}}{\text{Time w/o precise interrupts}}$$

Performance Comparison

Blocking Stores

Number of Entries	In-order Completion	Reorder Buffer	Reorder w/ Bypass
3	1.23	1.33	1.31
4	1.23	1.22	1.17
5	1.23	1.20	1.14
8	1.23	1.18	1.12
10	1.23	1.18	1.12

Non-blocking Stores

Number of Entries	In-order Completion	Reorder Buffer	Reorder w/ Bypass
3	1.16	1.31	1.28
4	1.16	1.17	1.12
5	1.16	1.13	1.05
8	1.16	1.12	1.03
10	1.16	1.12	1.03

Dynamic Branch Prediction

Basic idea

Have hardware try to guess ***whether*** and ***where*** a branch instruction will go

Example

```
address  instrn
-------  ------
0x64     BNEZ R1, +4
 ...     ...
0x74     ADD R3, R2, R1
```

Start with Branch PC (0x64) and produce

Prediction (T==taken, NT==not taken)

Prediction + Target PC (0x74)

Prediction + Target PC + Target Instrn. (`ADD R3, R2, R1`)

Branch Prediction Buffer

Start with Branch PC (0x64) and produce

Prediction (T==taken, NT==not taken)

A.k.a. branch history table

Example Implementation: 4K x 1 memory

Read prediction with LSBs of branch PC

Change bit on misprediction

May use bit from wrong PC

Better predictors

Two-bit state machine (for loops)

Correlating predictors

Only helps if Target PC is known before branch resolved (e.g., not in DLX)

Branch Target Buffer

Start with Branch PC (0x64) and produce

Prediction + Target PC (0x74)

Example Implementation: A Special Cache

Index & tags: Branch PC

Data: prediction info & Target PC

Next "IF" stage can use Target PC

No pipeline bubble on correct predictions!

Considerations

Many more bits per entry than branch prediction buffer

Size & associativity

Store not-taken branches?

Branch Target Cache

Start with Branch PC (0x64) and produce

- Prediction + Target PC + Target Instrn.
  (`ADD R3, R2, R1`)

Example Implementation: A Special Cache

- Index & tags: Branch PC
- Data: prediction info & Target PC
  & one or more target instrns

Could send Target Instrn rather than branch into the ID stage

- Zero-cycle branch: "branch folding"

Considerations

- Many more bits per entry than
  branch target buffer
- Size & associativity

Dynamic Branch Prediction Summary

Considerations

- Always correct
  (e.g., squash mispredicted instrns)
- Don't slow cycle time (much)
- Very fast on correct predictions
- High prediction accuracy
- Not too slow on incorrect predictions

Bottom line

- Useful for single-issue pipelines
- Critical for multiple-issue machines
  (coming soon) where the number of cycles
  to the next branch is often one or two!