Lecture 9: Dynamic Scheduling

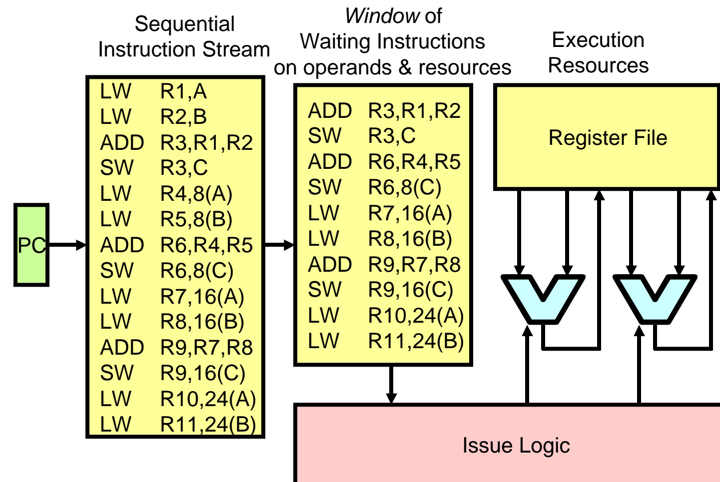
Kunle Olukotun
Gates 302
kunle@ogun.stanford.edu

http://www-leland.stanford.edu/class/ee282h/

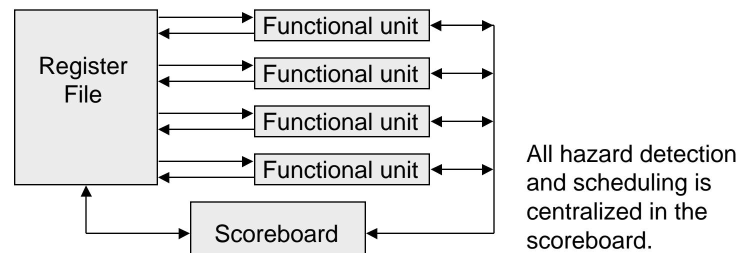
Hardware schemes for ILP

- Why HW at runtime?
  - » works when compiler can't know dependencies or latencies (e.g. cache miss)
  - » compiler is simpler
  - » code is less pipeline dependent
- In our pipeline so far, if an instruction stalls, everything after it stalls too ("in-order execution"):
  - »
```
divd   fp0, fp2, fp4      ;this takes a long time
addd   fp10, fp0, fp8     ;and this must stall for it
subd   fp8, fp8, fp14     ;but why couldn't this go ahead?
```
- We want to be able to *decode* instructions into the pipe in order but then let them stall individually while waiting for operands before *issue* to execution units.
- Dynamic scheduling (out-of-order issue/execution)

Dynamic Scheduling Basic Concept



Scoreboarding: The big picture



All hazard detection and scheduling is centralized in the scoreboard.

- The pipeline now has an extra stage (even ignoring load/stores), plus possible wait cycles for clearance from the scoreboard
  - » IF: instruction fetch
  - » ID: instruction decode. Stall the pipe if structural or WAW hazard.
  - » SB: scoreboard: wait for any RAW hazards to clear
  - » RF: read operands from the register file
  - » EX: execute in a functional unit,
  - » WB: wait for any WAR hazard to clear, then write back to register file

Scoreboarding: Pipelined example

```
multd   fp0, fp2, fp4     ;4 cycles
sd      fp0, 0(r4)        ;2 cycles
addu    r4, r2, #8        ;2 cycles
neg     fp0, fp4          ;1 cycle
```

	1	2	3	4	5	6	7	8	9	10	11	12	
multd	IF	ID	SB	RF	EX	EX	EX	EX	WB				
sd		IF	ID	SB	SB	SB	SB	SB	SB	RF	EX	WB	
addu			IF	ID	SB	RF	EX	EX	--	--	WB		
neg				IF	ID	ID	ID	ID	ID	SB	RF	EX	WB

Scoreboarding: The data structures

Instruction status

Instruction	Issued?	Operands read?	Execution complete?	Result written?

FU status

Functional unit	Busy?	Op	Dest reg	Source 1 reg	Source 1 from FU	Source 1 ready?	Source 2 reg	Source 2 from FU	Source 2 ready?

Result registers pending

	FP0	FP2	FP4	FP6	FP8	FP10	...	FP30
Result will come from which FU?								

Scoreboarding:
The rules

- Issue
  - » IF: FU not busy (no structural hazards) and no result pending for destination register (no WAW hazards)
  - » THEN: Assign FU entry.
    - – “From FU” comes from entry in “register result pending”
    - – “Ready?” (which means “ready but not yet read”) is TRUE only if there is no register result pending for that register
- Register file
  - » IF: Source1 and Source2 are both “ready” (no RAW hazards)
  - » THEN: ready=FALSE for both, read registers, and start EX
- Writeback
  - » IF: None of the functional units needs to use our result register and has yet to read it (no WAR hazards)
  - » THEN: Make “ready” any functional unit sources waiting for this FU, write the register file, clear the FU status, and clear our destination’s register result pending entry.

Scoreboarding:
A detailed example

Instruction status

Instruction	Issued?	Operands read?	Execution complete?	Result written?
ld f6, ...	y	y	y	y
ld f2, ...	y	y	y	
multd f0,f2,f4	y			
subd f8, f6, f2	y			
divd f10, f0, f6	y			
addd f6, f8, f2				

FU status

Functional unit	Busy?	Op	Dest reg	Source 1 reg	Source 1 from FU	Source 1 ready?	Source 2 reg	Source 2 from FU	Source 2 ready?
Integer	y	load	f2	f3		n	--		
Mult 1	y	mult	f0	f2	integer	n	f4		y
Mult 2	n								
Add	y	sub	f8	f6		y	f2	Integer	n
Divide	y	div	f10	f0	Mult 1	n	f6		y

Result registers pending

	FP0	FP2	FP4	FP6	FP8	FP10	...	FP30
Result will come from which FU?	Mult 1	Integer			Add	Divide		

Scoreboarding: Summary

- + Permits out-of-order execution and out-of-order completion for non-data-dependent operations
- + Centralizes decision making -- about as big as one functional unit in the 6600
- - Requires lots of busses to/from register file and functional units
- - No forwarding: all operands come from the register file
  - » But since results are written as soon as possible (without waiting through MEM, etc.), the benefit of adding forwarding is only one cycle
- - Stalls for all WAR and WAW hazards

Reducing WAR and WAR hazards: Compiler (static) register renaming

- Consider the fragment

```
divd    fp1, fp2, fp3    ;this will take a long time
subd    fp2, fp3, fp4    ;this can't start until divd is finished
                         ;with fp2
std     fp2, 0(r1)
```

- There is a WAR antidependence on fp2 between the divd and the subd.
- But by "renaming" fp2 and subsequent uses of it we avoid the hazard:

```
divd    fp1, fp2, fp3    ;this will take a long time
subd    fp5, fp3, fp4    ;but now this can start anytime
std     fp5, 0(r1)
```

- Reduces stalls for some pipelines, but uses additional registers

Register renaming again: this time in hardware!

- The trick: Every time you write to a register number, allocate a new physical register. Think of it as multiple copies of each register, one for each new value written to it.
- Implementation implications:
  - » Need more physical registers than register addresses
  - » Keep a pool of unassigned physical registers, and allocate one when a register is written
  - » All uses to the original register number must be remapped to the new physical register
  - » When a register is written, all previous physical registers allocated for it can be freed as soon as all previous instructions which use it have completed
- The big win over scoreboarding: no stalls for WAR and WAW hazards!

Dynamic register renaming: an example

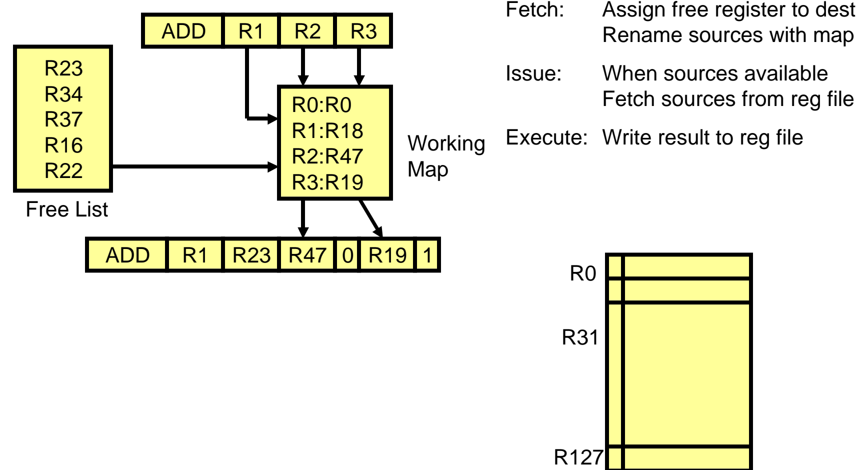
- Original code

```
multd   fp0, fp2, fp4
sd      fp0, 0(r4)
addu    r4, r2, #8
addd    fp0, fp6, fp8
```

- Two WAR hazards: r4 (sd/addu), and fp0 (sd/addd)
- Renamed code

  multd $fp0_a$, $fp2_a$, $fp4_a$
  sd $fp0_a$, 0($r4_a$)
  addu $r4_b$, $r2_a$, #8
  addd $fp0_b$, $fp6_a$, $fp8_a$

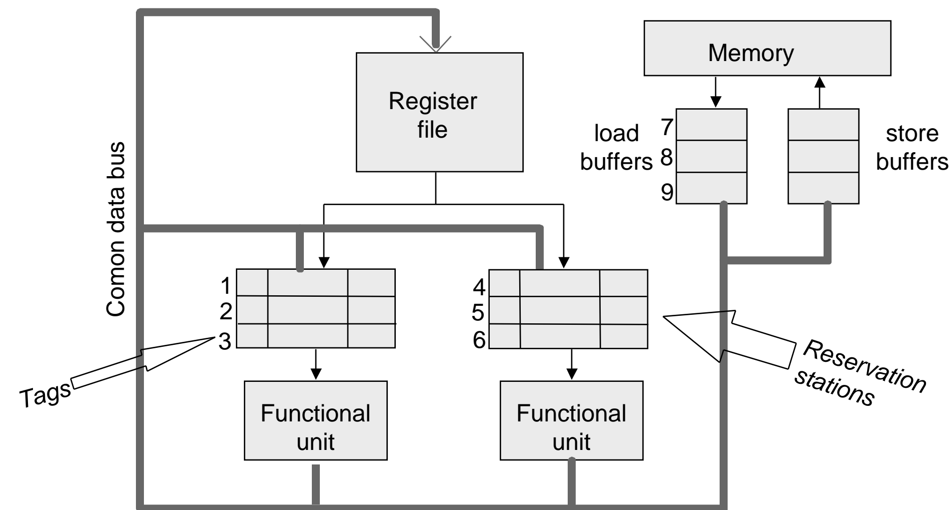
Register Renaming
Use additional *physical* registers



Tomasulo's Algorithm:
The big picture

- Replace the "functional unit status" table with "reservation stations" (distributed window):
  - » Each function unit has a set of reservations stations for instructions waiting to use it
  - » The reservation station stores operand *values* as they become available, **or** the name of a reservation station that will provide the result. Thus the original register number is no longer relevant, and is now "renamed" as a reservation station
- Each result is broadcast to the Common Data Bus (CDB) and can be snagged by any reservation station that needs it.
- Instructions can now be issued even if there are structural hazards or WAR/WAW hazards.

Tomasulo's Algorithm: Hardware Structure



Tomasulo's Algorithm: The pipeline stages

- Pipeline stages
  - » IF: instruction fetch
  - » ID: instruction decode
    - – reservation station assignment (may stall)
    - – operand "fetch"
  - » RS: reservation station, until RAW hazards clear
  - » EX: execute, arbitrate for CDB
  - » WB: write results to CDB (and register file)
- How data hazards are eliminated:
  - » RAW: delay in reservation station until operands ready
  - » WAR: fetch operands early and in issue order
  - » WAW: only last writer (in issue order) will update the register file

Tomasulo's algorithm: Pipeline example

```
multd   fp0, fp2, fp4   ;4 cycles
sd      fp0, 0(r4)      ;2 cycles
addu    r4, r2, #8      ;2 cycles
ld      fp0, 0(r5)      ;2 cycles
```

	1	2	3	4	5	6	7	8	9	10	11	12
multd	IF	ID	RS	EX	EX	EX	EX	WB				
sd		IF	ID	RS	RS	RS	RS	RS	EX	EX	WB	
addu			IF	ID	RS	EX	EX	--	WB			
ld				IF	ID	RS	EX	EX	--	WB		

CDB stalls

Tomasulo's Algorithm: The data structures

Reservation stations

Functional unit	Busy?	Op	Source 1 value	Source 1 tag	Source 2 value	Source 2 tag
Add1						
Add2						
Add3						
Mult1						xx
Mult2						

Reservation stn # or load buffer #

store buffers

busy?	tag	addr/value

load buffers

	busy?	addr
1		
2		
3		

Five functional units with one reservation stn each.

Register status

	FP0	FP2	FP4	FP6	FP8	FP10	...	FP30
Result will come from which res'vn stn								

Tomasulo's Algorithm: The rules

- Issue
  - » IF: a reservation station is available
  - » THEN: Assign a reservation station entry
    - – If a source register has no resulting pending. then read it's value from the file; otherwise set the tag to the station from which the result will come.
    - – Set the register status for our result register to our station # (Note that it may override something already there!)
- Execute
  - » IF: source1 and source2 values are present
  - » THEN: begin execution
- Writeback
  - » IF: CDB slot is available
  - » THEN:
    - – Write to all registers for which we're the destination, and zero the register status.
    - – In all reservation stations or store buffers that list us as an operand tag, write the value and clear the tag.

Tomasulo and Memory Operations

- Maintain ordering with respect to stores
  - » A load or store cannot execute before (after) a preceding (following) store unless they are known to be to different addresses
  - » Memory disambiguation
  - » hard at compile time, easy at run time

Tomasulo's Algorithm: An example

Reservation stations

Functional unit	Busy?	Op	Source 1 value	Source 1 tag	Source 2 value	Source 2 tag
Add1	y	sub	xxxxxxxx			Load2
Add2	y	add		Add1		Load2
Add3	n					
Mult1	y	mul		Load2	yyyyyyyy	
Mult2	y	div		Mult1	zzzzzzzz	

store buffers

busy?	tag	addr/value

load buffers

	busy?	addr
1	n	
2	y	xxx
3	n	

Instruction	Issued?	Exec?	WB?
ld f6, xxx	y	y	y
ld f2, xxx	y	y	
multd f0, f2, f4	y		
subd f8, f6, f2	y		
divd f10, f0, f6	y		
addd f6, f8, f2	y		

Register status

	FP0	FP2	FP4	FP6	FP8	FP10	...	FP30
Result will come from which res'vn stn	Mult1	Load2		Add2	Add1	Mult2		

Tomasulo's Algorithm: A big winner in loops

- If we can handle branches well, Tomasulo's algorithm allows parallel execution of multiple iterations of a loop body just like compiler loop unrolling.
- ; for (i=0; i<100; ++i) a[i] = a[i] * c;

```
loop:   ld      f0, 0(r1)
        multd   f4, f0, f2
        sd      0(r1), f4
        subi    r1, r1, #8
        bnez    r1, loop
```

With two multd units, two iterations of this loop will issue and execute in parallel.

Tomasulo's Algorithm: Notes and Summary

- Still need a contention strategy for:
  - » Access to the CDB
  - » Access to functional units with more than one reservation station
  - » Combination of FIFO and priority?
- Need to check for memory hazards: loads must check the addresses in the store buffers ("dynamic memory disambiguation")
- Advantages
  - » Increases parallelism by dynamic scheduling
  - » Eliminates WAR and WAW hazards by dynamic register renaming
  - » Expands the available registers without affecting programs or compilers
- Disadvantages:
  - » Lots of associative (parallel) comparisons
  - » Imprecise interrupts (need reorder buffer or equivalent)
  - » Branch hazards are still a problem

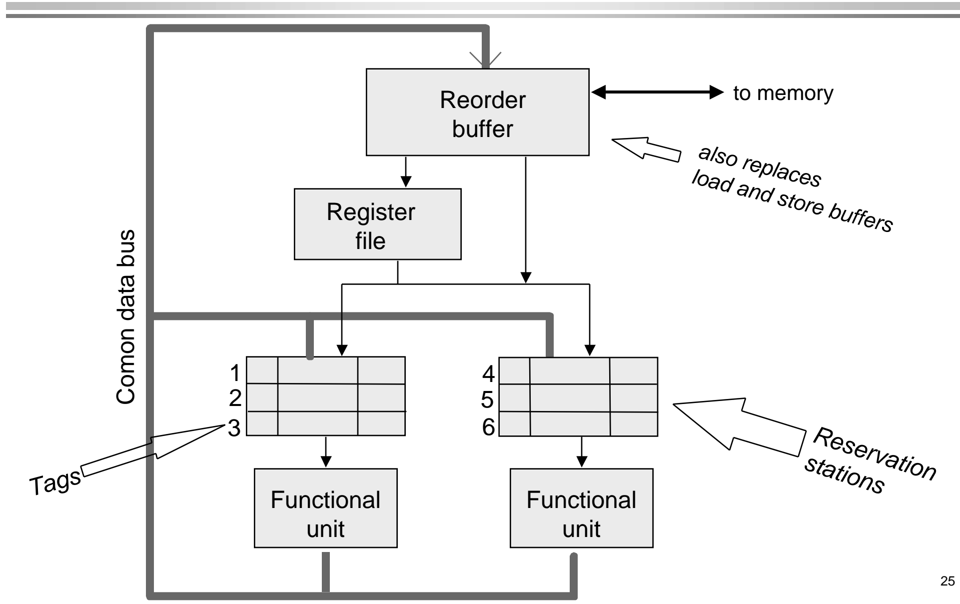
Extracting more parallelism: Speculation

- ILP is difficult to find in some common loops :
  - » for (p=head; p!=NIL; p = p->link) ++p->value;

```
        j       test
loop:   lw      r5, 0(r4)
        addi    r5, r5, #1
        sw      r5, 0(r4)
        lw      r4, 4(r4)
test:   bnez    r4, loop
```

- So far, we have predicted conditional branches and fetched based on the prediction. Now: actually *execute* the predicted instructions conditionally, and be prepared to nullify them.
- Speculative execution works well combined with dynamic scheduling, especially Tomasulo's algorithm
  - » Issue in-order, execute out-of-order, *commit* in-order
  - » Combining reservation stations with a reorder buffer gives us speculative execution **and** precise interrupts!

Tomasulo's algorithm with speculation



Tomasulo's Algorithm with speculation: The data structures

Reservation stations

Functional unit	Busy?	Op	Dest	Source 1 value	Source 1 tag	Source 2 value	Source 2 tag
Add1	n						
Add2	n						
Add3	n						
Mult1	n	mult	#3	yyyyyyyy		ssssssss	
Mult2	y	div	#5		#3	xxxxxxxx	

Reorder buffer

entry	busy?	Instr	State	dest	value
1	n	ld f6, 34(r2)	commit	f6	xxxxxxxx
2	n	ld f6, 45(r3)	commit	f2	yyyyyyyy
3	n	multd f0, f2, f4	writeback	f0	zzzzzzzz
4	y	subd f8, f6,f2	writeback	f8	wwwwww
5	y	divd f10, f0, f6	execute	f10	
6	y	addd f6, f8, f2	writeback	f6	uuuuuuuu

Register status

	FP0	FP2	FP4	FP6	FP8	FP10	...	FP30
Result will be in which reorder buffer?	#3			#6	#4	#5		

Tomasulo's Algorithm with speculation: The rules

- Issue
  - » IF: reservation station and reorder buffer slot is available
  - » THEN: queue for execution
    - – fill in the reservation station entry (copying operands from reorder buffer or register file)
    - – fill in the reorder buffer entry, state = "execute"
- Execute
  - » WHEN source1 and source2 values are present or broadcast on CDB
  - » THEN begin execution
- Writeback
  - » WHEN: CDB slot is available
  - » THEN: Free the reservation station
    - – Write to all reorder buffer slots for which we're the destination
    - – In all reservation stations that list our buffer slot as an operand tag, write the value and clear the tag
- Commit
  - » WHEN we reach the top of the reorder buffer and have a value
  - » THEN:
    - – Write the register (or store into memory)
    - – Zero the register status and free the reorder buffer entry

Branches and Exceptions

- When a branch occurs
  - » Make a reorder buffer entry for "branch"
  - » Predict the direction and fetch/execute
- When the branch "commits" (reaches the top of the reorder buffer)
  - » If the prediction was wrong, clear the reorder buffer and start fetching from the other path
- Works great for correct prediction, but misprediction penalty is high
  - » Might partially clear the reorder buffer and restart fetch as soon as the branch condition is resolved, rather than waiting for it to be at the top
  - » Even more exotic: speculatively execute both directions! (Many machines fetch both directions, but few execute.)
- Exceptions:
  - » Record exceptions in the reorder buffer, and don't recognize it until it's at the top. Interrupts are precise, and the "register status" array takes the place of the future file.