In-order vs. Out-of-order Execution

In-order instruction execution

- instructions are issued in compiler-generated order
- one stalls, they all stall
- instructions are statically scheduled by the compiler, statically scheduled or issued by the hardware

Out-of-order instruction execution

- instructions issued when a functional unit is available (no structural hazards)
- instructions executed when their operands are have been calculated or loaded from memory (no data hazards)
- can go around stalled instructions that are waiting for their operands to be computed
- instructions are dynamically scheduled by the hardware
- instructions are fetched in-order
- instruction completion may be in-order or out-of-order

Dynamic Scheduling

Flow dependences:

- flow dependences are upheld
  - consumer not issued until its operands have been produced
  - a producer and its consumers execute in program order

program order (in-order execution)

`lw $3, 100($4)`	in execution, cache miss
`add $2, $3, $4`	waits until the miss is satisfied
`sub $5, $6, $7`	waits for the add

execution order (out-of-order execution)

`lw $3, 100($4)`	in execution, cache miss
`sub $5, $6, $7`	in execution
`add $2, $3, $4`	waits until the miss is satisfied -- flow dependence still upheld

Out-of-Order Execution

Several implementations

- **out-of-order completion**
    - CDC 6600 with **scoreboarding**
    - IBM 360/91 with **Tomasulo's algorithm & reservation stations**
    - out-of-order leads to:.
        - imprecise interrupts
        - WAR hazards
        - WAW hazards
- in-**order completion**
    - MIPS R10000 (M) with **register renaming**
    - Intel Pentium Pro (X) with the **reorder buffer**

Scoreboarding

Scoreboarding (CDC 6600)

- technique for allowing out-of-order execution
- centralized hardware
- detects structural/data hazards
- monitors all hazard detection & resolution:
    - when source operands can be read (RAW)
    - when result can be written (WAW, WAR)
        - stalls the dependent instruction

Scoreboarding

Scoreboard functions

issue	structural & WAW data hazard detection in-order issue if no hazards
read	RAW hazard detection read registers & out-of-order execution if no hazard
execute	scoreboard is updated that execution has completed
write	WAR hazard detection stalls if a hazard allows register write if no hazard

Scoreboard hardware

- state
  - instruction status for each issued or pending instruction
  - FU status for each functional unit
  - result register status for each register
- control
  - hazard detection
  - issue logic
  - bookkeeping logic

Step through the example in the book!

Tomasulo's Algorithm

Tomasulo's Algorithm (IBM 360/91)

- out-or-order execution capability plus register renaming

- **reservation stations**:
  - buffers for functional units that hold instructions stalled for RAW hazards
  - **register renaming** to eliminate WAR & WAW hazards
    - buffer operands as soon as available
    - subsequent instructions refer to reservation station locations, not registers
    - last writer updates the register
  - more reservation stations than registers, so eliminates more false dependences than a compiler can

- distributed hazard detection & execution control
  - reservation stations at each functional unit

- **common data bus** ships all results to all reservation stations that are waiting

- **dynamic memory disambiguation**
  - loads associatively check addresses in store buffer

Tomasulo's Algorithm

Tomasulo functions

issue & read — structural hazard detection for reservations stations & load/store buffers
read registers if operand in them
register renaming to eliminate WAR & WAW hazards

execute — RAW hazard detection
snoop on common data bus for missing operands
dispatch to functional unit

write — broadcast result on common data bus to registers & reservation stations

Tomasulo state

- operation of the issued instructions waiting for execution
- tag in reservation stations, registers, store buffer
  - names of "virtual" registers used in renaming
  - reservation station & load buffer entries
    - which reservation station will produce the operand
    - which load buffer contains the operand
- operand values
- busy fields
- addresses in load/store buffers (for memory disambiguation)

Step through the example in the book

Tomasulo's Algorithm

Tag in the reservation station/store buffer/register file indicates where the result will come from

Handling WAR hazards

`ld F1,_`

`addf _, F1,_` source will come from the load buffer

`addf F1,_` will write to F1

not matter if `ld` finish last

Handling WAW hazards

`addf F1,F0,F8` F1's tag will *originally* say `addf`'s reservation station

...

`subf F1,F8,F14` F1's tag will *now* say `subf`'s reservation station

no register will claim the `addf` result if completes last

Tomasulo's Algorithm

Dynamic loop unrolling

- F0 field in reservation station says where value comes from
- `ADDF` and `SD` in each iteration has a different tag
- only the last iteration writes to F0

```
LOOP:  LD F0, 0(R1)
       ADDF F0, F0, F1
       SD 0(R1), F0
       SUBT R1, R1, #8
       BNEZ R1, LOOP
```

Nice features relative to static loop unrolling

- effectively increases number of registers -- no register pressure
- dynamic memory disambiguation
- simpler compiler

Downside

- loop control instructions still executed

Comparison of Scoreboarding & Tomasulo

Differences between them:

- hazard detection & execution control
  distributed (Tomasulo) vs.
  centralized (scoreboarding)
- handling WAW & WAR hazards
  hardware register renaming eliminates (Tomasulo) vs.
  explicit hazard detection & stalling (scoreboarding)
- handling RAW hazards
  results forwarded to FUs: available immediately (Tomasulo)
  vs. write/read registers (scoreboarding)
- writing results
  forwarding to *all* FUs *simultaneously* (Tomasulo) vs.
  trunk lines (scoreboarding)
- path to registers
  1 common data bus (Tomasulo) vs.
  multiple buses to registers -- the big hardware cost
  (scoreboarding)

Dynamic Scheduling

Advantages over static scheduling

- more effective at exploiting parallelism
  - increased instruction throughput
  - increased functional unit utilization
- more registers to work with
- makes dispatch decisions dynamically, based on when instructions actually complete & operands are available
- can *completely* disambiguate memory references
- efficient execution of code compiled for a different pipeline
- simpler compiler in theory

Disadvantages in the 1960s

- hardware complexity (a biggie!)
- small scheduling window
- imprecise interrupts

Use both!

Dynamic Scheduling

Why go out of style?

- expensive hardware for the time (actually, still is)
- register files grew so less register pressure
- early RISCs had lower CPIs

Why come back?

- higher chip densities
- multiple instruction issue
- greater need to hide latencies as
  - discrepancy between CPU & memory speeds increases
  - branch misprediction penalty increases

What R10000 Register Renaming Does

Register renaming provides a **mapping** between 2 register sets

- **architectural registers** defined by the ISA
- **physical registers** implemented in the CPU
  - results of the instructions committed so far
  - results of subsequent, independent instructions that have completed execution
    - ~ issue width * # pipeline stages between register renaming & commit
- hazards determined by comparing physical register numbers

Effects:

- eliminates WAW and WAR hazards
- increases ILP

A Register Renaming Example

Code Segment	Register Mapping	Comments
`ld r7,0(r6)`	`r7 -> p1`	**p1** is allocated
`...`		
`add r8, r9, r7`	`r8 -> p2`	use **p1**, not r7
`...`		
`sub r7, r2, r3`	`r7 -> p3`	**p3** is allocated **p1** is deallocated when `sub` commits

R10000 Register Renaming Implementation

Modular design with regular hardware data structures

- 64 **physical registers** (integer & FP)
- **map table** for the *current* architectural-to-physical register mapping (integer & FP)
  - accessed with an architectural register number
  - produces a physical register number
  - new physical register number comes from a **free register list** (integer & FP)
- **instruction queue** (integer, FP & data transfer)
  - contains decoded & mapped instructions with the current physical register mappings
  - determines when operands are available
    - compares each source operand to destinations being written this cycle
  - determines when an appropriate functional unit is available
  - dispatches instructions to functional units
    - removes instructions from the queue
  - instructions entered to free locations

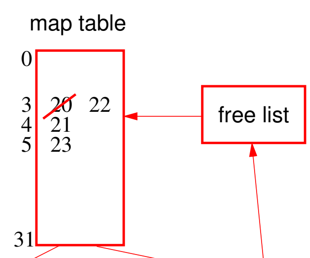
R10000 Register Renaming Implementation

- one **active list** for all uncommitted instructions
  - instructions in program-generated order
  - instructions removed from the active list when:
    - an instruction commits
    - branch is mispredicted
    - an exception occurs
  - contains the *previous* physical destination register mapping
    - released to the free list when the instruction commits
    - used to recreate the map table for instruction restart after an exception
  - instructions in the other hardware structures & the functional units are identified by their active list location
- **busy-register table** (integer & FP):
  - indicates whether a physical register contains a value
  - bit is set when a register is mapped & leaves the free list (works like a full/empty bit or a reservation bit)
  - cleared when a FU writes the register
  - used to determine operand availability

Comparators to find dependences (on architectural registers) between instructions being decoded in parallel

The R10000 Register Renaming Process

```
ld      A3, #(reg)      def, multi-cycle
add     A4, A3, reg     use
sub     A3, reg, reg    redef, independent
or      A5, reg, A3     independent
```



Instruction Queue				
Ins	S1	Dest	AL tag	Busy
ld	u	P20	1	00
add	P20	P21	2	10
sub	u	P22	3	00
or	P22	P23	4	00

	Activity List		
	Dest	Arch	Done bit
ld	u	A3	not done
add	u	A4	not done
sub	P20	A3	done
or	u	A5	done

R10000 Execution

In-order issue
- rename registers via a map table
- detect structural hazards for instruction queues & active list
- issue instructions

Out-of-order execution (to increase ILP)
- reservation-station-like instruction queues that indicate when an operand has been calculated
- detect functional unit structural & RAW hazards
- dispatch instructions

In-order completion (to preserve precise interrupts)
- this & previous program-generated instructions have completed
- physical register in previous mapping returned to free list
- rollback on interrupts

Pentium Pro Register Renaming Implementation

Register renaming

- important because very few GPRs
- **reorder buffer** *(~ R10K active list)*
  - holds 40 **uops** in program order
    (instructions are decoded into RISClike uops)
  - field for either integer or FP uop result
    - 3 specific results can be written each cycle
    - arbitrate if a structural hazard for the reorder buffer
- **register alias table** *(~ R10K map table)*
  - indicates whether a source operand points to the reorder buffer or the real register file
    (real register file ~ IBM 360/91 physical registers)
  - associative search to do the data hazard check

Out-of-order execution

- **reservation station** *(~ R10K instruction queue)*
  - result values go back to the reservation station so dependent uops have source operand values
    (but looks more like IBM 360/91 reservation stations)

Pentium Pro Execution

In-order issue

- enter instructions into reorder buffer for in-order completion
- rename registers via register alias table
- detect structural hazards for reservation station

Out-of-order execution

- one reservation station
- RAW hazards checked
- separate integer, FP, memory units
- result goes to reservation station & reorder buffer

In-order completion

- this & previous instructions have completed
- write “G”PR register
- rollback on interrupts

Limits

Limits on out-of-order execution

- amount of ILP in the code
- scheduling window size
- number & types of functional units
- number of renaming registers (M P IBM 360/91) or
  number of WAW & WAR hazards (CDC)
- cycle time