18-640 Foundations of Computer Architecture

Lecture 6:
"Register Data Flow Techniques"

John Paul Shen
September 11, 2014.

- Required Reading Assignment:
  - **Chapter 5 of Shen and Lipasti (SnL).**
- Recommended References:
  - Robert Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of Research and Development, 11(1):25-33, January 1967.
  - Bo Zhang, "A Tomasulo's Algorithm Emulator," February 1, 2013.

Electrical & Computer ENGINEERING
Carnegie Mellon University

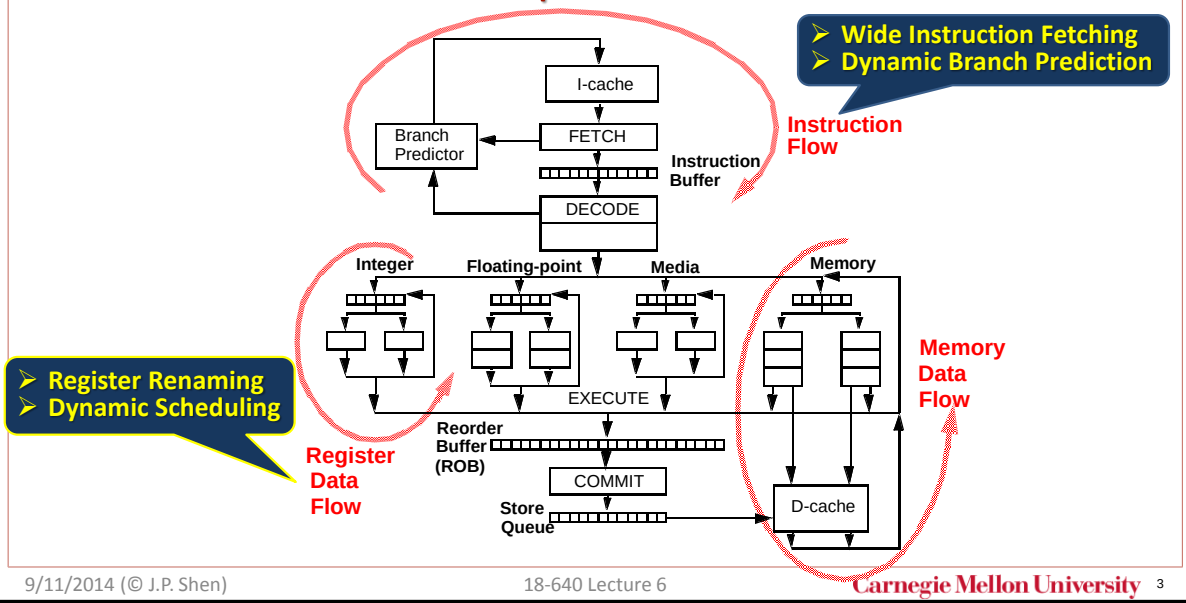
18-640 Foundations of Computer Architecture

Lecture 6:
"Register Data Flow Techniques"

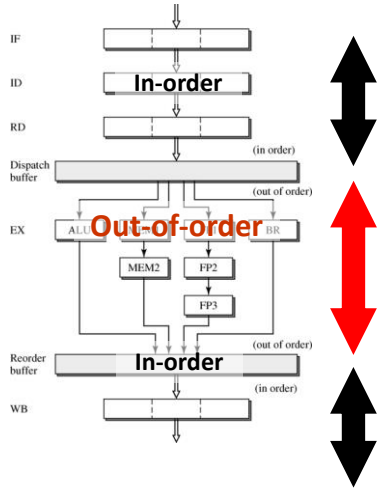
A. Register Data Flow
B. Register Renaming
C. Tomasulo's Algorithm

Electrical & Computer ENGINEERING
Carnegie Mellon University

Three Flow Paths of Superscalar Processors



Now Let's Move On to "Register Data Flow"



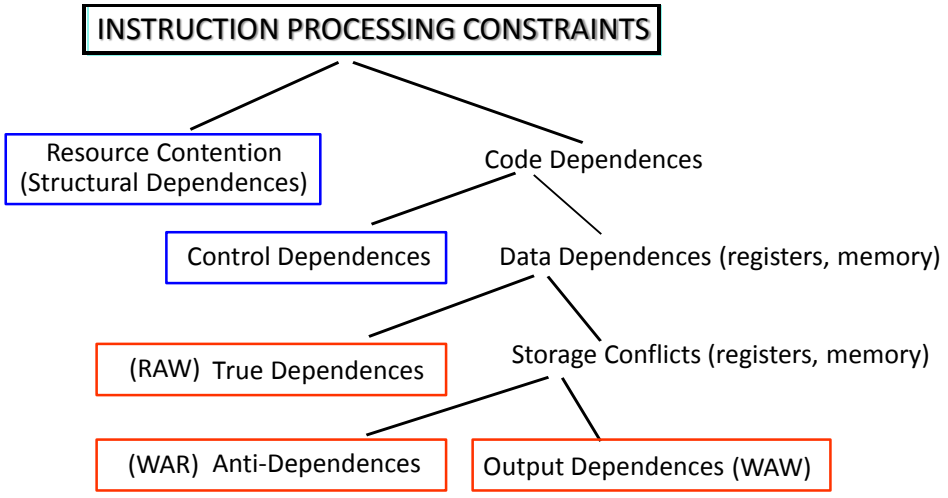
- We have: fetched & decoded instructions
  - In-order but speculative (branch prediction)
- Register Renaming
  - Address limitation of score-boarding:
    - One pending instruction per destination register
  - Eliminate WAR and WAW dependences without stalling
- Dynamic Scheduling
  - Track & resolve true RAW dependences
  - Scheduling HW: Instruction window, reservation stations, common data bus, …

A. Register Data Flow (ALU Penalty)

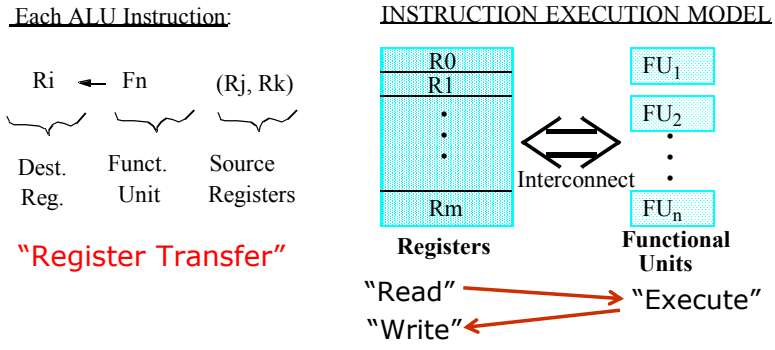
a. Resolving Anti-dependences
b. Resolving Output Dependences
c. Resolving True Data Dependences

Electrical & Computer ENGINEERING

The Big Picture: Impediments Limiting ILP



Register Data Flow



- For an instruction to execute:
  - Need availability of functional unit Fn (structural dependence)
  - Need availability of Rj and Rk (RAW: true data dependence)
  - Need availability of Ri (WAR and WAW: anti and output dependences)

Causes of Register Storage Conflict

REGISTER RECYCLING

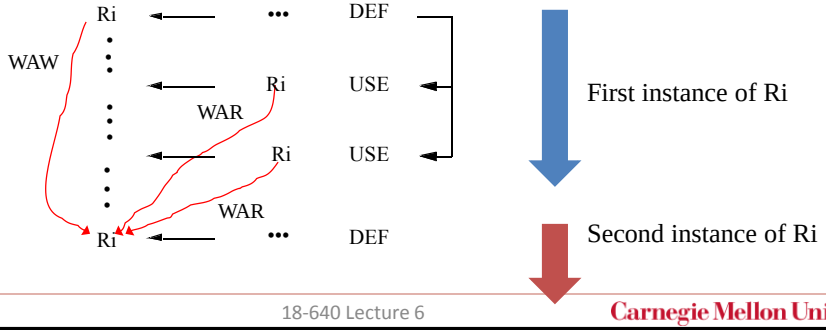
MAXIMIZE USE OF REGISTERS

MULTIPLE ASSIGNMENTS OF VALUES TO REGISTERS

OUT OF ORDER ISSUING AND COMPLETION

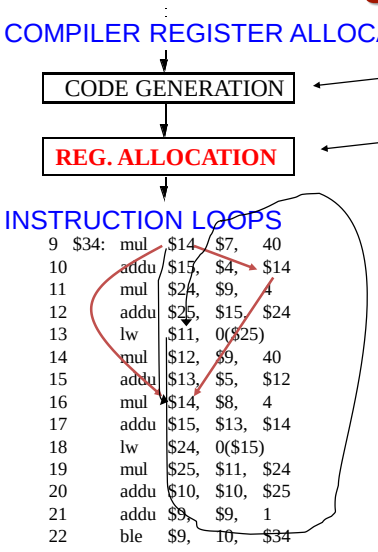
LOSE IMPLIED PRECEDENCE OF SEQUENTIAL CODE

LOSE 1-1 CORRESPONDENCE BETWEEN VALUES AND REGISTERS



Reason for WAW and WAR: Register Recycling

- Intermediate code
  - Infinite number of symbolic registers
  - One used per value definition
- Register Allocation via graph coloring
  - Map symbolic registers to few architectural registers
  - Leads to register reuses



Single Assignment, Symbolic Reg.

Map Symbolic Reg. to Physical Reg.
Maximize Reuse of Reg.

"Spill code" (if not enough registers)

```
For (k=1;k<= 10; k++)
   t += a [i] [k] * b [k] [j] ;
```

- Dynamic register reuse
  - Reuse same set of registers in each iteration
  - Overlapped execution of multiple iterations

Resolving Anti-Dependences

```
    .
    .
    .
(1) R4  ←  R3 + 1
(2) R3  ←  R5 + 1
```

Must Prevent (2) from completing before (1) is dispatched.

STALL DISPATCHING

DELAY DISPATCHING OF (2)

REQUIRE RECHECKING AND REACCESSING

WAR only — **COPY OPERAND**

COPY NOT-YET-USED OPERAND TO PREVENT BEING OVERWRITTEN

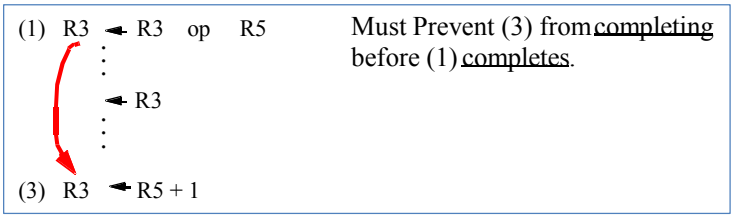
MUST USE TAG IF ACTUAL OPERAND NOT-YET-AVAILABLE

WAR and WAW — **RENAME REGISTER**

HARDWARE ALLOCATION

```
R3   ←  …
     ←  R3
R3'  ←  …
     ←  R3'
```

Resolving Output Dependences



STALL DISPATCHING/ISSUING

DENOTE OUTPUT DEPENDENCE

HOLD DISPATCHING UNTIL RESOLUTION OF DEPENDENCE

ALLOW DECODING OF SUBSEQUENT INSTRUCTIONS

RENAME REGISTER

HARDWARE ALLOCATION

```
R3   ← …
     ← R3
R3'  ← …
     ← R3'
```

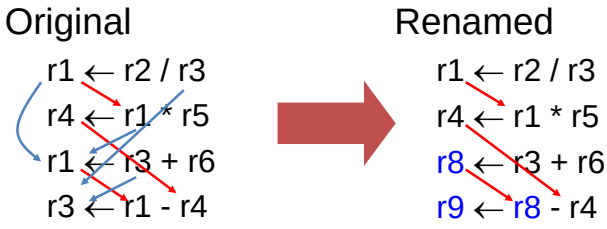

B. Register Renaming

Register Renaming: The Idea

- Anti and output dependences are false dependences

$r_3 \leftarrow r_1$ op r_2
$r_5 \leftarrow r_3$ op r_4
$r_3 \leftarrow r_6$ op r_7

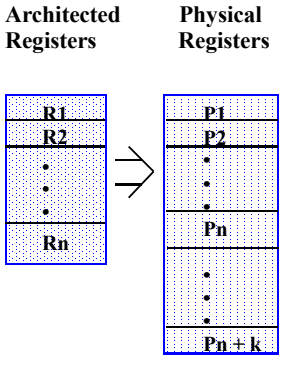
- The dependence is on name/location rather than data
- Given unlimited number of registers, anti and output dependences can always be eliminated



Register Renaming

Register Renaming Resolves:

Anti-Dependences
Output Dependences



Design of Redundant Registers

Number:
- **One**
- **Multiple**

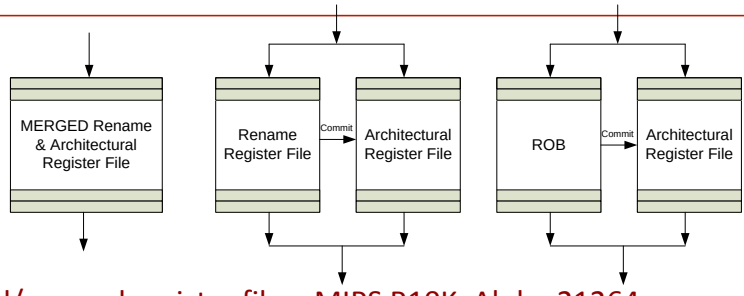
Allocation:
- **Fixed for Each Register**
- **Pooled for all Regsiters**

Location:
- **Attached to Register File (Centralized)**
- **Attached to functional units (Distributed)**

Register Renaming Implementation

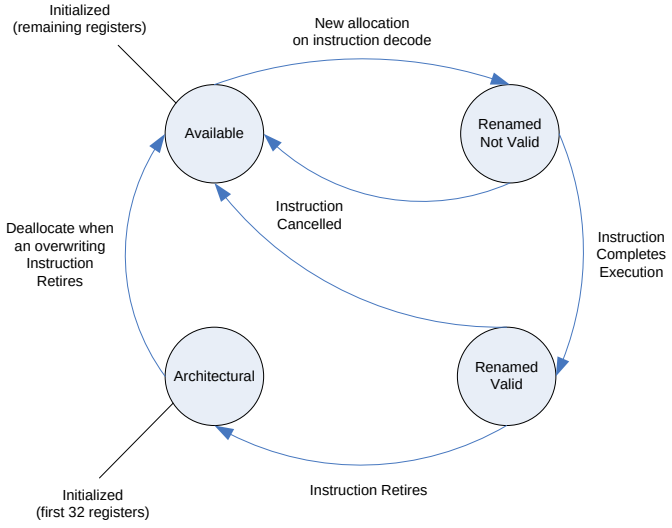
- Renaming:
  - Map small set of architecture registers to a large set of physical registers
  - New mapping for architectural register when it is assigned a new value
- Renaming buffer organization (how are registers stored)
  - Unified RF, split RF, renaming in the ROB
  - RF = register file
- Number of renaming registers
- Number of read/write ports
- Register mapping (how do I find the register I am looking for)
  - Allocation, de-allocation, and tracking

Renaming Buffer Options



- Unified/merged register file – MIPS R10K, Alpha 21264
  - Registers change role architecture to renamed
- Rename register file (RRF) – PA 8500, PPC 620
  - Holds new values until they are committed to ARF (extra transfer)
- Renaming in the ROB – Pentium III
- Note: can have a single scheme or separate for integer/FP

Unified Register File: Physical Register FSM



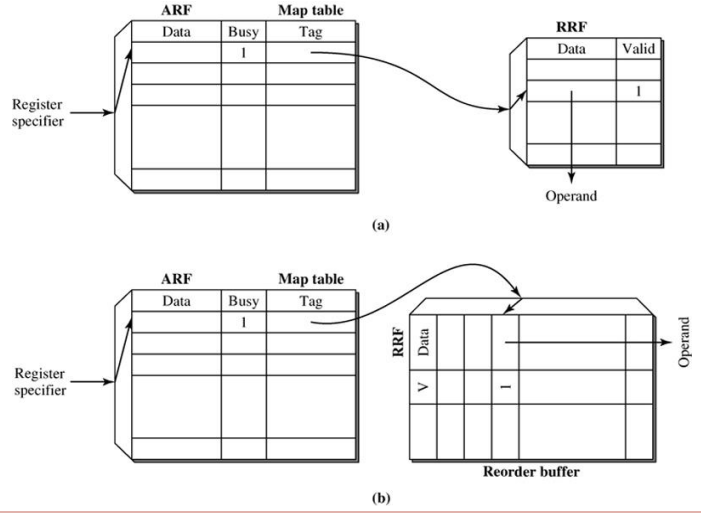
Number of Rename Registers

- Naïve: as many as the number of pending instructions
  - Waiting to be scheduled + executing + waiting to commit
- Simplification
  - Do not need renaming for stores, branches, …
- Usual approach:
  - # scheduler entries ≤ # RRF entries ≤ # ROB entries
- Examples:
  - PPC 620: scheduler 15, RRF 16 (RRF), ROB 16
  - MIPS R12000: scheduler 48, RRF 64 (merged), ROB 48
  - Pentium III: scheduler 20, RRF 40 (in ROB), ROB 40

Register File Ports

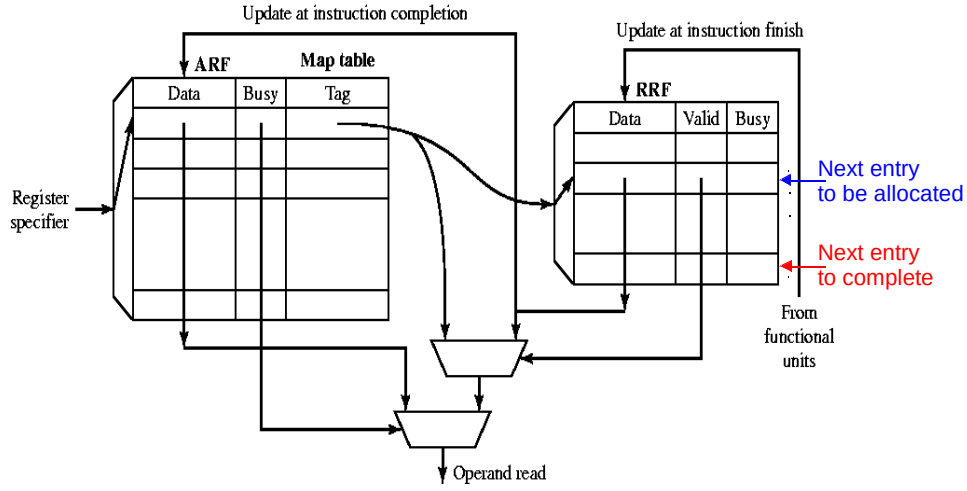
- Read: if operands read as instructions enter scheduler
  - Max # ports = 2 * # instructions dispatched
- Read: if operands read as instruction leave scheduler
  - Max #ports = 2* # instructions issued
- Write: # of FUs or # of instructions committing
  - Depends on unified vs separate rename registers
- Notes:
  - Can implement less ports and have structural hazards
    - Need control logic for port assignment & hazard handling
  - When using separate RRF and ARF, need ports for the final transfer
  - Alternatives to increasing ports: duplicated RF or banked RF
    - What are the issues?

Integrating Map Tables with the ARF



Register Renaming Tasks

- Source Read, Destination Allocate, Register Update

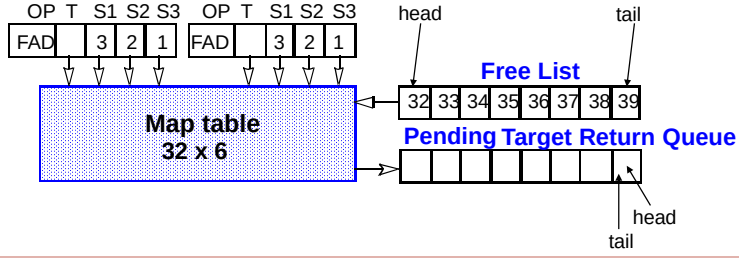


Register Renaming in the IBM RS/6000

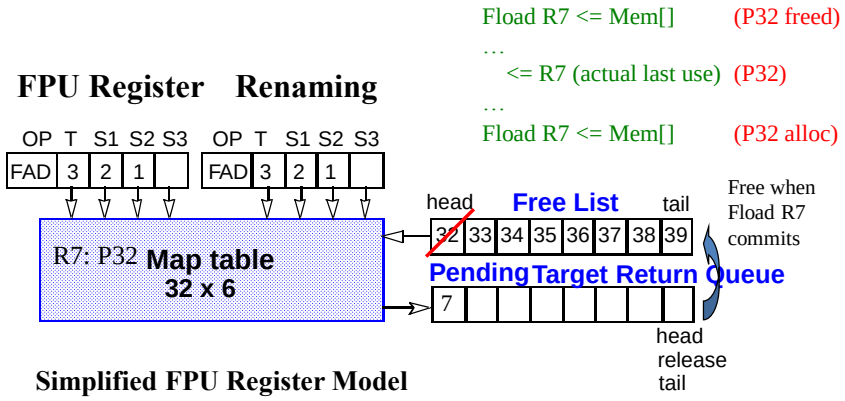
Incoming FPU instructions pass through a renaming table prior to decode
Physical register names only within the FPU!!

32 architectural registers ⇒ 40 physical registers
Complex control logic maintains active register mapping

FPU Register Renaming



Register Renaming in the IBM RS/6000 FPU



Simplified FPU Register Model

Incoming FPU instructions pass through a renaming table prior to decode

The 32 architectural registers are remapped to 40 physical registers

- Physical register names are used within the FPU
- Complex control logic maintains active register mapping

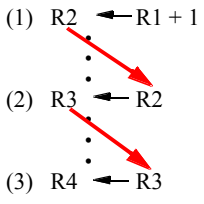
Renaming Difficulties: Wide Instruction Issue

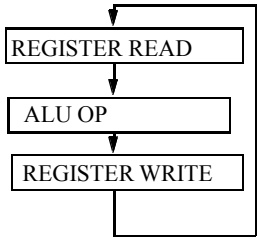
- Need many ports in RFs and mapping tables
- Instruction dependences during dispatching/issuing/committing
  - Must handle dependences across instructions
  - E.g. add R1←R2+R3; sub R6←R1+R5
  - Implementation: use comparators, multiplexors, counters
    - Comparators: discover RAW dependences
    - Multiplexors: generate right physical address (old or new allocation)
    - Counters: determine number of physical registers allocated

Renaming Difficulties: Mispredictions & Exceptions

- On exception/misprediction, register mapping must be precise
- Separate RRF: consider all RRF entries free
- ROB renaming: consider all ROB entries free
- Unified RF: restore precise mapping
  - Single map: traverse ROB to undo mapping (history file approach)
    - ROB must remember old mapping…
  - Two maps: architectural and future register map
    - On exception, copy architectural map into future map…
  - Checkpointing: keep regular check points of map, restore when needed
    - When do we make a checkpoint? On every instruction? On every branch?
    - What are the trade-offs?
    - We'll revisit this approach later on…

Resolving True Data Dependences





STALL DISPATCHING
ADVANCE INSTRUCTIONS
"DYNAMIC EXECUTION"

Reservation Station + Complex Forwarding
Out-of-order (OoO) Execution
Try to Approach the "Data-Flow Limit"

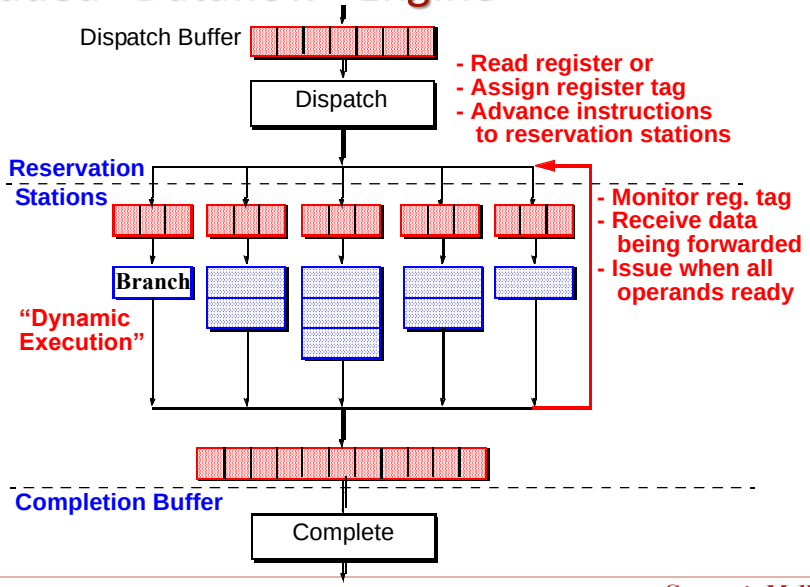
1) Read register(s), get "IOU" if not ready
2) Advance to reservation station
3) Wait for "IOU" to show up
4) Execute

Register Renaming and Dynamic Scheduling

- Register Renaming: address limitations of the scoreboard
  - Scoreboard limitation
    - Up to one pending instruction per destination register
  - Eliminate WAR and WAW dependences without stalling

- Dynamic scheduling
  - Track & resolve true-data dependences (RAW)
  - Scheduling hardware:
    - Instruction window, reservation stations, common data bus, …
  - Original proposal: Tomasulo's algorithm [Tomasulo, 1967]

Embedded "Dataflow" Engine

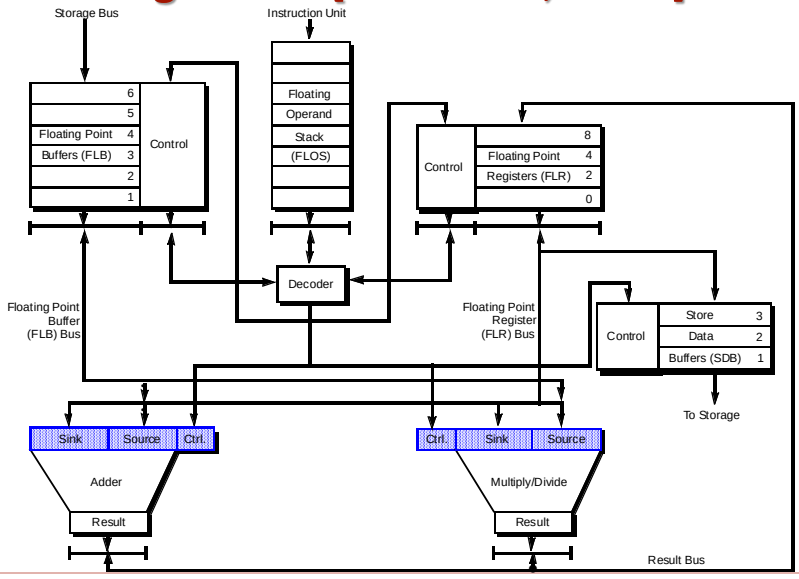


C. Tomasulo's Algorithm

a. Modified IBM 360/91 Floating-point Unit
b. Reservation Stations
c. Common Data Bus
d. Register Tags
e. Operation of Dependence Mechanisms

Electrical & Computer ENGINEERING

Tomasulo's Algorithm [Tomasulo, 1967]



IBM 360/91 FPU

- **Multiple functional units (FU's)**
  - Floating-point add
  - Floating-point multiply/divide
- **Three register files (pseudo reg-reg machine in floating-point unit)**
  - (4) floating-point registers (FLR)
  - (6) floating-point buffers (FLB)
  - (3) store data buffers (SDB)
- **Out of order instruction execution**:
  - After decode the instruction unit passes all floating point instructions (in order) to the floating-point operation stack (FLOS) [actually a queue, not a stack]
  - In the floating point unit, instructions are then further decoded and issued from the FLOS to the two FU's
- **Variable operation latencies**:
  - Floating-point add: 2 cycles
  - Floating-point multiply: 3 cycles
  - Floating-point divide: 12 cycles
- Goal: **achieve concurrent execution of multiple floating-point instructions, in addition to achieving one instruction per cycle in instruction pipeline**

Dependence Mechanisms

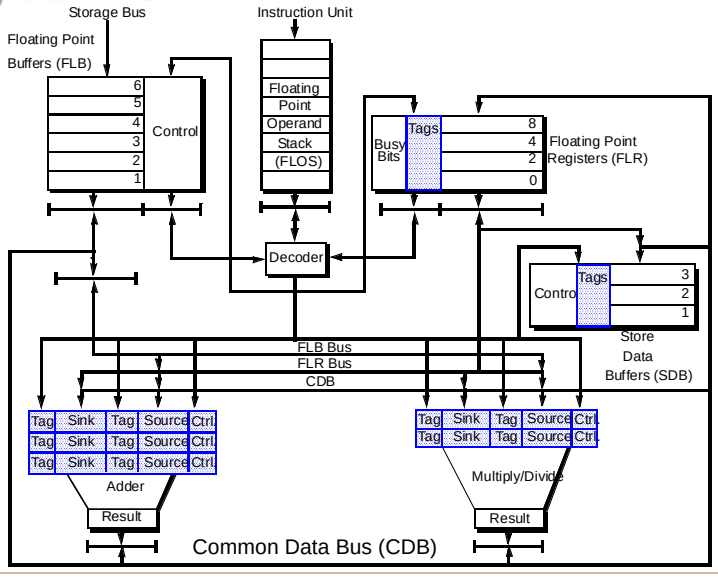
Two Address IBM 360 Instruction Format:

R1 <-- R1 op R2

Major dependence mechanisms:

- **Structural (FU) dependence = > virtual FU's**
  - Reservation stations
- **True dependence = > pseudo operands + result forwarding**
  - Register tags
  - Reservation stations
  - Common data bus (CDB)
- **Anti-dependence = > operand copying**
  - Reservation stations
- **Output dependence = > register renaming + result forwarding**
  - Register tags
  - Reservation stations
  - Common data bus (CDB)

IBM 360/91 FPU



Reservation Stations

- Used to collect operands or pseudo operands (tags).
- Associate more than one set of buffering registers (control, source, sink) with each FU, → virtual FU's.
- Add unit: three reservation stations
- Multiply/divide unit: two reservation stations

Tag	Sink	Tag	Source
0 implies valid data	Source value	0 implies valid data	Source value

Common Data Bus (CDB)

- CDB is fed by all units that can alter a register (or supply register values) and it feeds all units which can have a register as an operand.
- Sources of CDB:
  - Floating-point buffers (FLB)
  - Two FU's (add unit and the multiply/divide unit)
  - 6 FLB + 3 addRS + 2 muldivRS = 11 unique sources
  - 3 physical sources (FLB, adder, mul/div)
- Destinations of CDB:
  - Reservation stations
  - Floating-point registers (FLR)
  - Store data buffers (SDB)
  - (5 RS x 2) + 4 FLR + 3 SDB : CDB has 17 destinations
- Electrically very challenging
  - 3 physical sources must arbitrate for access to CDB
  - Tag + data must be driven to 17 destinations

Register Tags

- **Every source of a register value must be uniquely identified by its own tag value.**
  - (6) FLB's
  - (5) reservation stations (3 with add unit, 2 with multiply/divide unit)

    ➔ 4-bit tag is needed to identify the 11 potential sources

- **Every destination of a register value must carry a tag field**.
  - (5) "sink" entries of the reservation stations
  - (5) "source" entries of the reservation stations
  - (4) FLR's
  - (3) SDB's

    ➔ a total of 17 tag fields are needed (i.e. 17 places that need tags)

Operation of Dependence Mechanisms

1. **Structural (FU) dependence ➔ virtual FU's**
   - FLOS can hold and decode up to 8 instructions.
   - Instructions are dispatched to the 5 reservation stations (virtual FU's) even though there are only two physical FU's.
   - Hence, structural dependence does not stall dispatching.
2. **True dependence ➔ pseudo operands + result forwarding**
   - If an operand is available in FLR, it is copied to a res. station entry.
   - If an operand is not available (i.e. there is pending write), then a tag is copied to the reservation station entry instead. This tag identifies the source of the pending write. This instruction then waits in its reservation station for the true dependence to be resolved.
   - When the operand is finally produced by the source (ID of source = tag value), this source unit asserts its ID, i.e. its tag value, on the CDB followed by broadcasting of the operand on the CDB.
   - All the reservation station entries and the FLR entries and SDB entries carrying this tag value in their tag fields will detect a match of tag values and latch in the broadcasted operand from the CDB.
   - Hence, true dependence does not block subsequent independent instructions and does not stall a physical FU. Forwarding also minimizes delay due to true dependence.

Example 1

i: R2 <= R0 + R4
j: R8 <= R0 + R2 (RAW on R2)

Cycle #1 — DISPATCHED INSTRUCTION(S): i

	ID	Tag	Sink	Tag	Source
i	1	0	6.0	0	10.0
	2				
	3				
	Adder	i			

	ID	Tag	Sink	Tag	Source
	4				
	5				
	Mult/Div				

	Busy	Tag	Data
0			6.0
2	x	1	3.5
4			10.0
8			7.8

Cycle #2 — DISPATCHED INSTRUCTION(S): j

	ID	Tag	Sink	Tag	Source
i	1	0	6.0	0	10.0
j	2	0	6.0	1	16.0
	3				
	Adder	i			

	ID	Tag	Sink	Tag	Source
	4				
	5				
	Mult/Div				

	Busy	Tag	Data
0			6.0
2	x	1	16.0
4			10.0
8	x	2	7.8

Cycle #3 — DISPATCHED INSTRUCTION(S):

	ID	Tag	Sink	Tag	Source
	1				
j	2	0	6.0	0	16.0
	3				
	Adder	j			

	ID	Tag	Sink	Tag	Source
	4				
	5				
	Mult/Div				

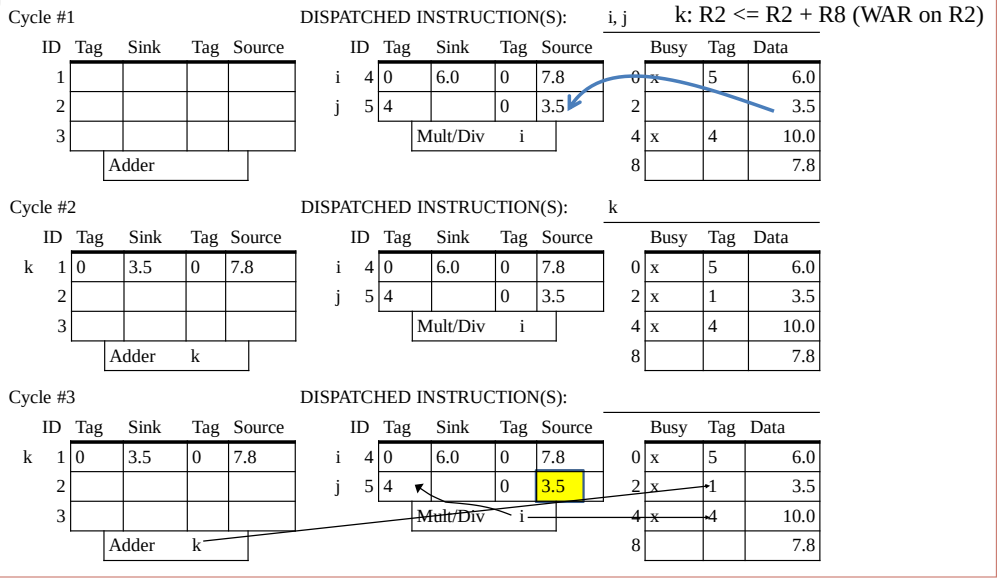
	Busy	Tag	Data
0			6.0
2			16.0
4			10.0
8	x	2	7.8

Operation of Dependence Mechanisms

3. **Anti-dependence ➔ operand copying**
   - If an operand is available in FLR, it is copied to a reservation station entry.
   - By copying this operand to the reservation station, all anti-dependences due to future writes to this same register are resolved.
   - Hence, the reading of an operand is not delayed, possibly due to other dependences, and subsequent writes are also not delayed.

Example 2

i: R4 <= R0 * R8
j: R0 <= R4 * R2 (RAW on R4)
k: R2 <= R2 + R8 (WAR on R2)



Operation of Dependence Mechanisms

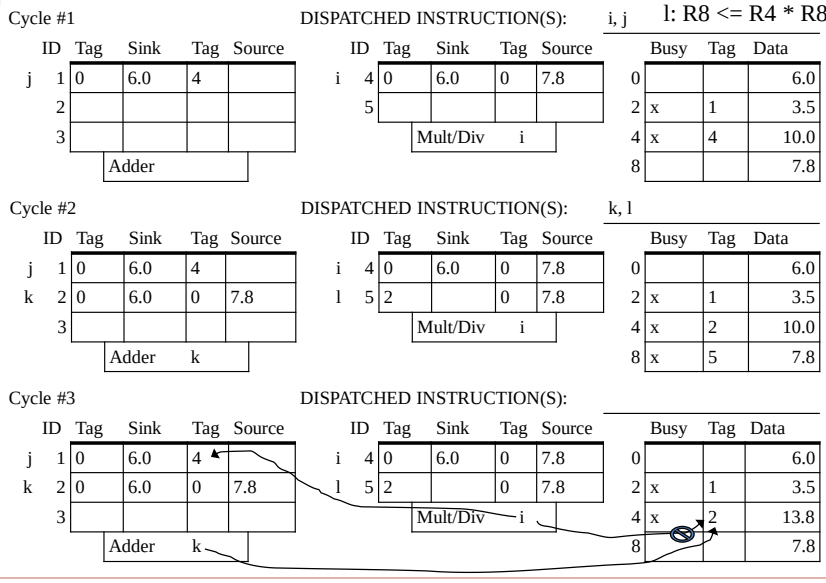
3. Output dependence ➔ register renaming + result forwarding

- If a register is waiting for a pending write, its tag field will contain the ID, or tag value, of the source for that pending write.
- When that source eventually produces the result, that result will be written into the register via the CDB.
- It is possible that prior to the completion of the pending write, another instruction can come along and also has that same register as its destination register.
- If this occurs, the operands (or pseudo operands) needed by this instruction are still copied to an available reservation station. In addition, the tag field of the destination register of this instruction is updated with the ID of this new reservation station, i.e. the old tag value is overwritten. This will ensure that the said register will get the latest value, i.e. the late completing earlier write cannot overwrite a later write.
- Hence, the output dependence is resolved without stalling a physical functional unit, not requiring additional buffers to ensure sequential write back to the register file.

Example 3

What if j causes FP overflow exception?
- where is R4?
- it is lost => imprecise exceptions!

i: R4 <= R0 * R8
j: R2 <= R0 + R4 (RAW on R4)
k: R4 <= R0 + R8 (WAW on R4)
l: R8 <= R4 * R8 (RAW on R4)



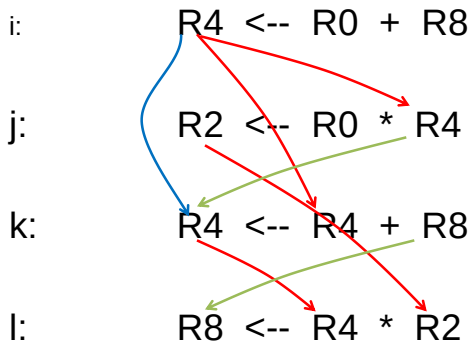
Summary of Tomasulo's Algorithm

- Supports out of order execution of instructions.
- Resolves dependences dynamically using hardware.
- Attempts to delay the resolution of dependences as late as possible.
- Structural dependence does not stall issuing; virtual FU's in the form of reservation stations are used.
- Output dependence does not stall issuing; copying of old tag to reservation station and updating of tag field of the register with pending write with the new tag.
- True dependence with a pending write operand does not stall the reading of operands; pseudo operand (tag) is copied to reservation station.
- Anti-dependence does not stall write back; earlier copying of operand awaiting read to the reservation station.
- Can support sequence of multiple output dependences.
- Forwarding from FU's to reservation stations bypasses the register file.

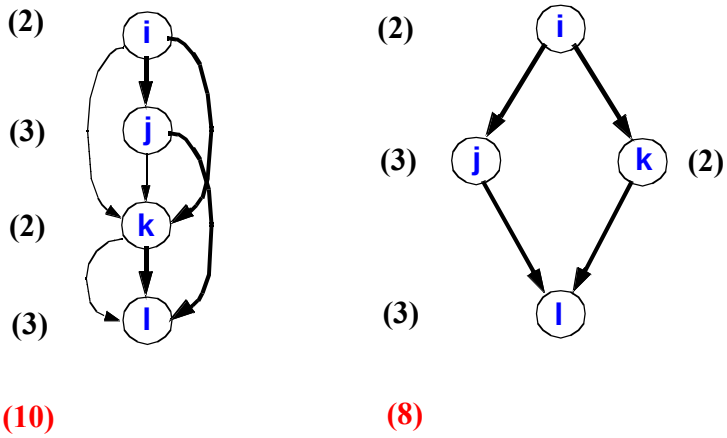
Tomasulo vs. Modern OOO

	IBM 360/91	Modern
Width	Peak IPC = 1	4+
Structural hazards	2 FPU Single CDB	Many FU Many busses
Anti-dependences	Operand copy	Reg. Renaming
Output dependences	Renamed reg. tag	Reg. renaming
True dependences	Tag-based forwarding	Tag-based forwarding
Exceptions	Imprecise	Precise (ROB)
Implementation	3 x 66" x 15" x 78" 60ns cycle time 11-12 gate delays per pipe stage >$1 million	1 chip 300ps < $100

Example 4



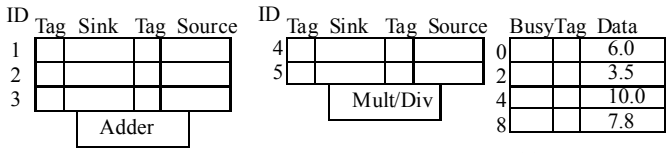
Example 4



Can Tomasulo's algorithm reach dataflow limit of 8?

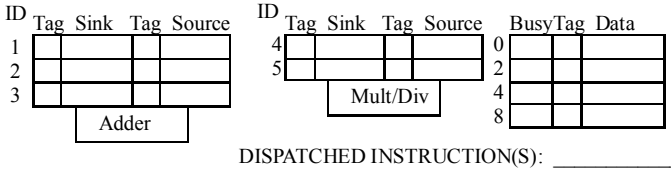
Example 4

CYCLE #1



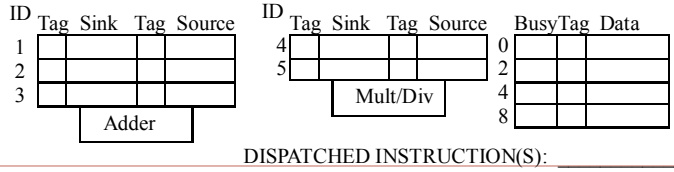
DISPATCHED INSTRUCTION(S): _____

CYCLE #2



DISPATCHED INSTRUCTION(S): _____

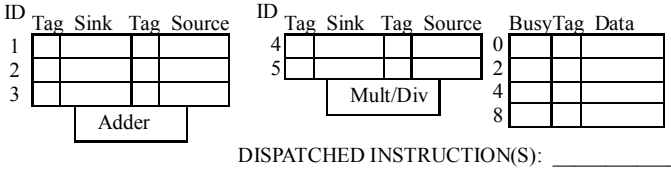
CYCLE #3



DISPATCHED INSTRUCTION(S): _____

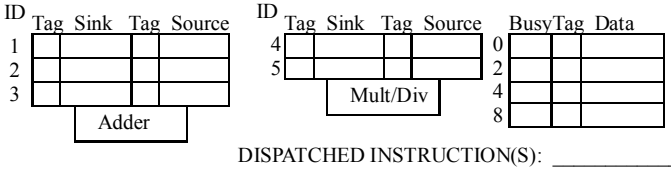
Example 4

CYCLE #4



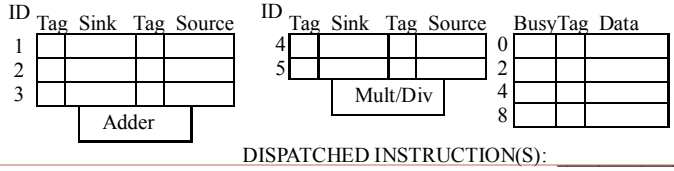
DISPATCHED INSTRUCTION(S): _____

CYCLE #5



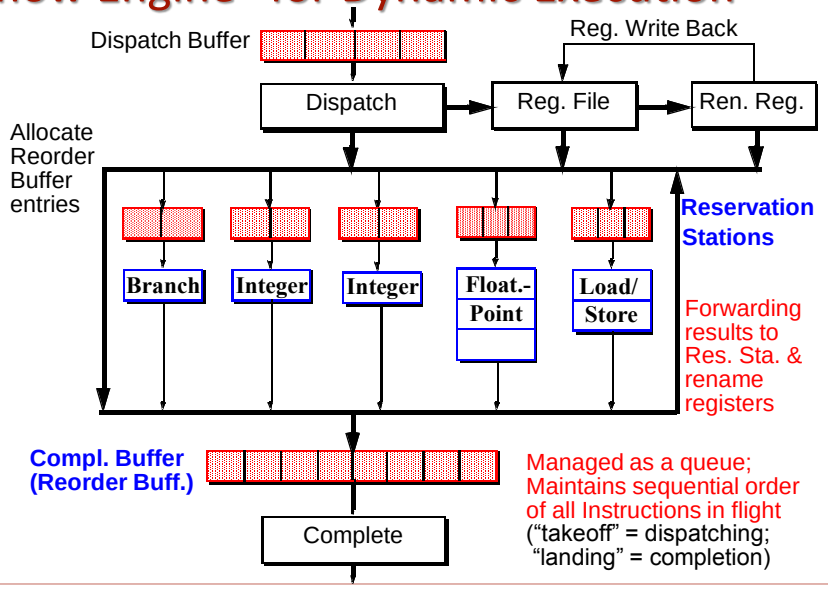
DISPATCHED INSTRUCTION(S): _____

CYCLE #6



DISPATCHED INSTRUCTION(S): _____

"Dataflow Engine" for Dynamic Execution



18-640 Foundations of Computer Architecture

Lecture 7: "Dynamic OOO Scheduling and Execution"

John Paul Shen

September 16, 2014

Next Time ...

- Required Reading Assignments:
  - **Chapters 5 and 7 of Shen and Lipasti (SnL).**
- Recommended Reference:
  - http://en.wikipedia.org/wiki/Pentium_Pro

Electrical & Computer ENGINEERING