EE 457 Unit 9a

Exploiting ILP

Out-of-Order Execution

Credits

- Some of the material in this presentation is taken from:
  - Computer Architecture: A Quantitative Approach
    - John Hennessy & David Patterson
- Some of the material in this presentation is derived from course notes and slides from
  - Prof. Michel Dubois (USC)
  - Prof. Murali Annavaram (USC)
  - Prof. David Patterson (UC Berkeley)

Exploiting Parallelism

- With increasing transistor budgets of modern processors (i.e., can do more things at the same time) the question becomes how do we find enough *useful* tasks to increase performance, or, put another way, what is the most effective way of exploiting parallelism!
- Many types of parallelism available
  - __________ **Level Parallelism (ILP)**: Overlapping instructions within a single process/thread of execution
  - __________ **Level Parallelism (TLP)**: Overlap execution of multiple processes/threads
  - __________ **Level Parallelism (DLP)**: Overlap an operation (instruction) that is to be applied independently to multiple data values (usually, an array)

```
for (int i=0; i < MAX; i++) { A[i] = A[i] + 5; }
```

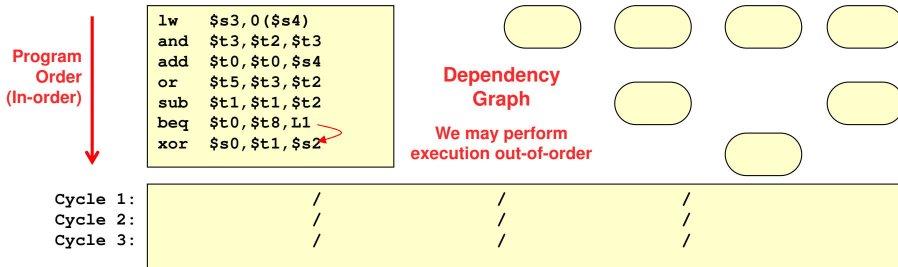
- We'll focus on **ILP** in this unit

Outline

- Instruction Level Parallelism
  - _______________ **(IO)** pipeline
    - From academic 5-stage pipeline
    - To 8-stage MIPS R4000 pipeline
    - Superscalar, superpipelined
  - _________________ **(OoO)** Execution
    - **This unit**: OoO Execution (Compute the result) AND OoO Completion (write result to memory or a register). (Problem: Exceptions)
    - **Next Unit**: OoO Execution BUT In-order completion

Instruction Level Parallelism (ILP)

- Although a program defines a sequential ordering of instructions, in reality many instructions can be executed in parallel (i.e. **out of (________) _____**).
- ILP refers to the process of finding instructions from a single program/thread of execution that can be executed in parallel
- Data flow (________________________) limits out-of-order execution
- _____________ instructions (no data dependencies) can be executed at the same time)
- _____________________ also provide some ordering constraints



Basic Blocks

- Basic Block (def.) = Sequence of instructions that will always be ______________________
  - No ______________________ out
  - No branch targets coming ____
  - Also called "straight-line" code
  - Average size: _____ instrucs.

```
     lw   $s3,0($s4)
     and  $t3,$t2,$t3
L1:  add  $t0,$t0,$s4
     or   $t5,$t3,$t2
     sub  $t1,$t1,$t2
     beq  $t0,$t8,L1
     xor  $s0,$t1,$s2
```

This is a basic block (starts w/ target, ends with branch)

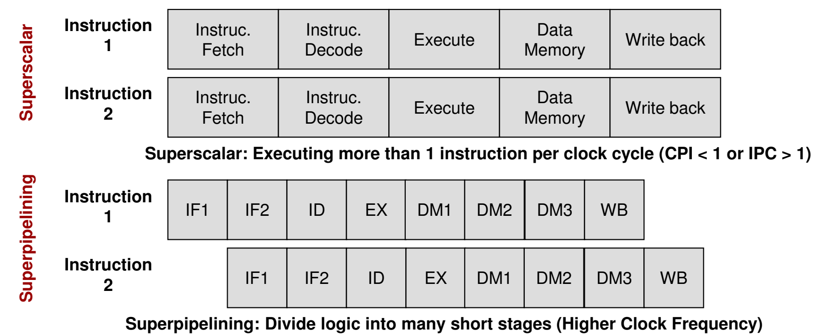
- Instructions in a basic block can be overlapped if there are no data dependencies
- _________ dependences really _________________ of possible instructions to overlap
  - W/o extra hardware, we can only overlap execution of instructions within a basic block

Other In-Order techniques

SUPERSCALAR & SUPERPIPELINING

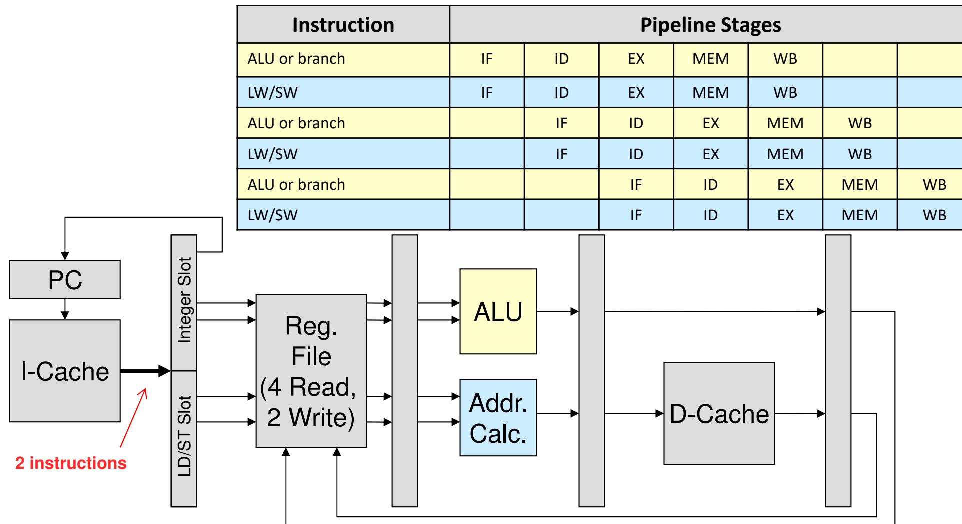
Overview

- Superscalar = More than 1 instruction completing ___________________
  - 2-way superscalar = Proc. that can issue 2 instructions per clock cycle
  - Success is sensitive to ability to find independent instructions to issue in the same cycle
- Superpipelining = Many small stages to boost _____________________
  - Success depends of finding instructions to schedule in the shadow of data and control hazards



2-way Superscalar

- Ex: One ALU & Data transfer (LW/SW) instruction can be issued at the same time
- Relies on compiler to find and reorder appropriate instructions (using nops if no appropriate instruction can be found



Instruction	Pipeline Stages						
ALU or branch	IF	ID	EX	MEM	WB		
LW/SW	IF	ID	EX	MEM	WB		
ALU or branch		IF	ID	EX	MEM	WB	
LW/SW		IF	ID	EX	MEM	WB	
ALU or branch			IF	ID	EX	MEM	WB
LW/SW			IF	ID	EX	MEM	WB

Sample Scheduling

- Compiler can reorder instructions to find integer and memory instructions to fuse together that can be run down the pipeline at the same time

```
void f1(int *A, int n) {
  do {
    *A += 5;
    A++;
    n--;
  } while (n != 0);
}
```

```
# $6 = A
# $7 = n = # of iterations
L1: ld   $9, 0(%6)
    add  $9, $9, 5
    st   %r9,0(%rdi)
    add  $6, $6, 4
    add  $7, $7, -1
    jne  $0,%esi,L1
```

time

Int./Branch Slot	LD/ST Slot
addi $7, $7, -1	lw $9,0($6)
addi $6, $6, 4	
addi $9, $9, 5	
bne $0,$7,L1	st $9,-4($6)

w/ modifications and code movement
IPC = 6 instrucs. / 4 cycle = 1.5

Scheduling Strategies

- ______________ Scheduling
  - _________ re-orders instructions in such a way that no dependencies will be violated and allows for OoOE
- ______________ Scheduling
  - ______ implementing the Tomasulo algorithm or other similar approach will re-order instructions to allow for OoOE
- More Advanced Concepts
  - Branch prediction and speculative execution (execution beyond a branch flushing if incorrect) will be covered later

Static Scheduling

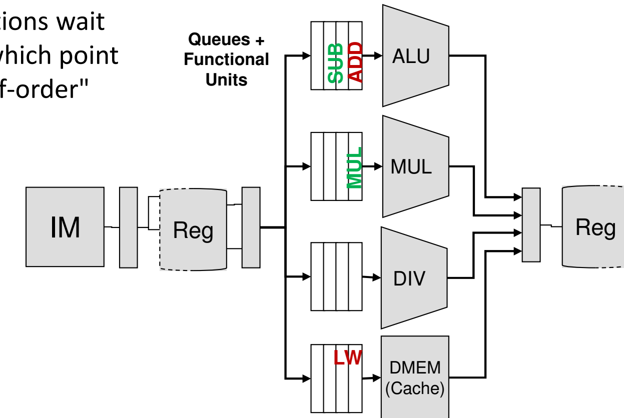
- Strengths
  - Hardware simplicity [Better clock rate]
    - Power/energy advantage
    - Compiler has a global view of the program anyway, so it should be able to do a “good” job
  - Very predictable: static performance predictions are reliable
- Weaknesses
  - Requires _______________ to take advantage of new/modified architecture
  - Cannot foresee dynamic (data-dependent) events
    - Cache miss, conditional branches (can only recedule instructions in a basic block)
  - Cannot precompute memory addresses
  - No good solution for precise exceptions with out-of-order completion

OUT-OF-ORDER EXECUTION

Out-of-Order Motivation

- We will focus on dynamically scheduled, OoO processors
- Hide the impact of dynamic events such as a ________________
  - Let _____________ instructions behind a stalled instruction execute
- Separate functional units (ALU, MUL, DMEM, etc.)
- "Queues" where instructions wait until they are ready at which point they can execute "out-of-order"

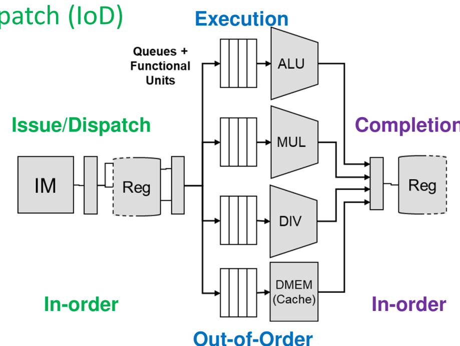
```
LW $4,0($5)
// cache miss
ADD $6,$7,$4
SUB $1,$2,$3
MUL $9,$7,$2
```



Dispatch, Execution, and Completion

- "Execution" here means ___________ the results not necessarily writing them to a register or memory
- Completion means committing/__________ the results to register file or memory
- While we say out-of-order execution we really mean/want:
  - In-order (Program order) Issue/Dispatch (IoD)
  - Out-of-Order Execution (OoOE)
  - In-order Completion (IoC) [hard]
    - So we'll start with the easier Out-of-Order Completion (OoOC)

```
LW $4,0($5)
// cache miss
ADD $6,$7,$4
SUB $1,$2,$3
MUL $9,$7,$2
```

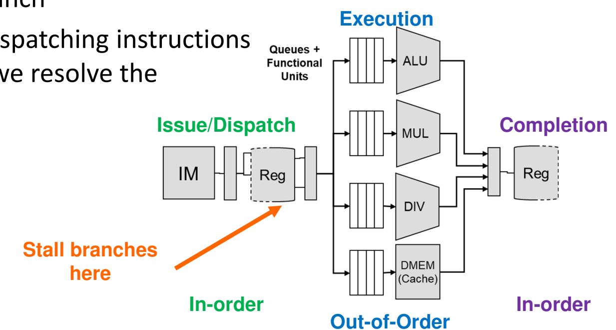


Branch Handling

- We will present the concept of OoOC (out-of-order completion) which is a bit easier and then come back to the desired approach of In-Order Completion (IOC)
- OoOC Issues
  - Branches…we should not commit an instruction that came after (in program order) a branch
  - Solution: _______ dispatching instructions after a branch until we resolve the outcome

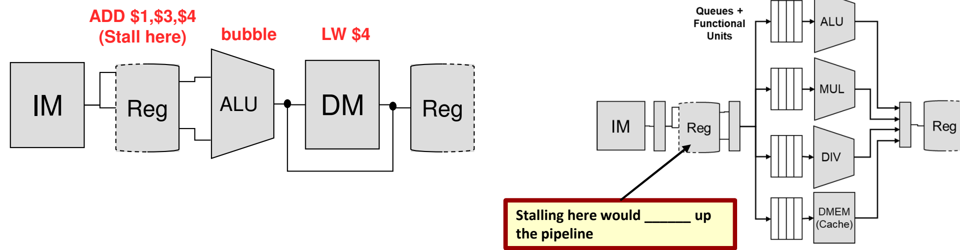
```
LW $4,0($5)   // cache miss
BEQ $4,$0,L1
ADD $6,$7,$8
// What if we execute this
ADD out of order
```

Stall branches here



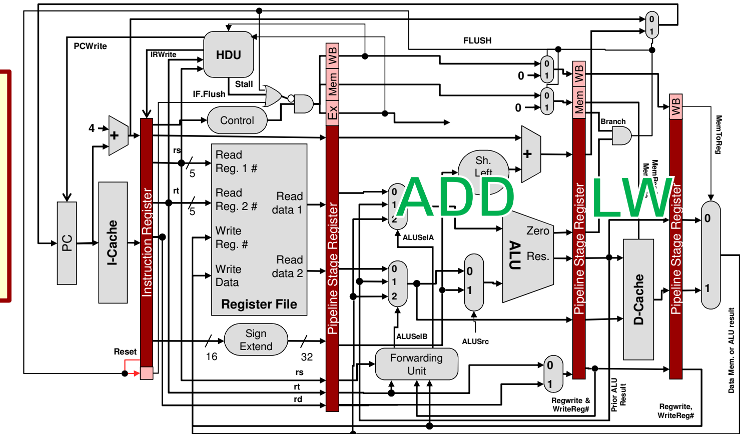
Data Hazard Stalling

- In our 5-stage pipeline (in-order execution) RAW dependency was solved by
  - Forwarding (preferably) or
  - Stalling (LW followed by dependent instruction)
- Dependent instructions stalled in the ID stage if necessary
- Do we want to stall in the decode stage in our OoO processor?
  - ____! Doing so would necessarily stall ______________________ us



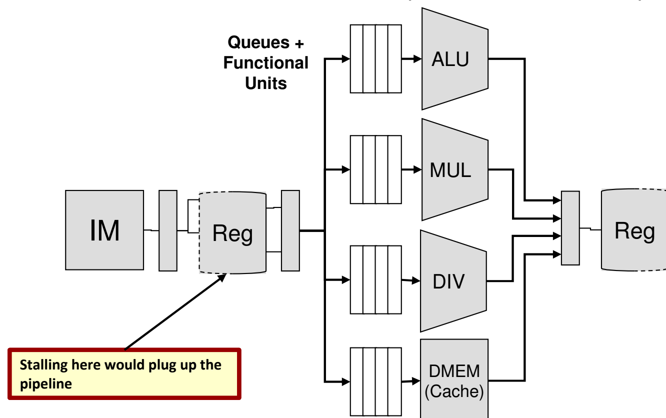
EX Stage Stalling

- In our 5-stage pipeline, could we have stalled in the EX stage
- ____! If ADD depended on an instruction in ____ then it has no place to ___________ that forwarded data while it stalls



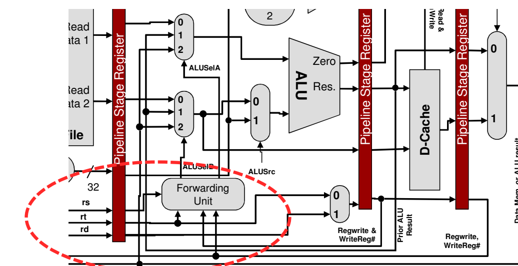
Where to Stall?

- But to implement OoO execution, we cannot stall in the decode stage since that would prevent any further issuing of instructions
- Thus, now we will issue to queues for each of the multiple functional units and have the instruction stall in the queue until it is ready



Forwarding in OoO Execution

- In 5-stage pipeline later instructions carried their source register IDs into the EX stage to be compared with destination register ID's of their earlier instructions
- But in OoO execution, we may have many (earlier) instructions in front of us and would require more complex hardware to determine who is producing the data we need (especially when multiple producers exist and we want the latest version)
- Instead, the dispatch unit will ______________ tell the dependent instruction who to get data from using part of Tomasulo's algorithm



Tomasulo's Plan

- OoO Execution
- Multiple functional units
  - Integer ALU, Data memory, Multiplier, Divider
- Queues between ID and EX stages (in place of ID/EX register)
  - Allows later instructions to keep issuing even if earlier ones are stalled
- Method for dealing with RAW data hazards by specifying who dependent instructions should get data from
  - But with OoO execution, _______________ arise!

WAR and WAW

NEW DATA HAZARDS

RAW, WAR, and WAW

- RAW = Read After Write
  - lw $8, 40($2)
  - add $9, $8, $7
- WAR = Write After Read
  - add $9, $8, $6 ← say $6 is not available yet, can LW execute?
  - lw $8, 40($2)
- WAW = Write After Write
  - add $9, $8, $6 ← say $6 is not available yet, can LW execute?
  - lw $9, 40($2)

Why would anyone produce one result in $9 without utilizing that result? Why would he overwrite it with another result? How is this possible?

WAW can easily occur

- How is WAW possible?
- Example 1
  - Say a company gives standard bonus to most of the employees and a higher bonus to managers
  - The software may set a default value to the standard bonus and then overwrite for the special case
- Example 2
  - Consider multiple iterations of a loop body

```
int x = standard_bonus;
if (manager)
  x = special_bonus;
set_bonus(x);
```

```
for(i=MAX; i != 0; i--)
  A[i] = A[i] * 3;
```

```
L1: lw   $2, 40($1)
    mult $4, $2, $3
    sw   $4, 40($1)
    addi $1, $1,-4
    bne  $1, $0,L1
```

Original Code

```
L1: lw   $2, 40($1)
    mult $4, $2, $3
    sw   $4, 40($1)
    addi $1, $1,-4
    bne  $1, $0,L1
L1: lw   $2, 40($1)
    mult $4, $2, $3
    sw   $4, 40($1)
    addi $1, $1,-4
    bne  $1, $0,L1
```

RAW, WAR, and WAW

- Some terminology to remember
- RAW = Read After Write
  - lw $8, 40($2)
  - add $9, $8, $7
- WAR = Write After Read
  - add $9, $8, $6
  - lw $8, 40($2)
- WAW = Write After Write
  - add $9, $8, $6
  - lw $9, 40($2)

Depdencies

RAW
A ______ dependency

WAR
An _____-dependency

WAW
An _____-dependency

Note: No information is communicated in WAR/WAW hazards.
If no info is communicated can we somehow solve these hazards?

RAW, WAR, and WAW

- In-order execution:
  - We need to deal with RAW only
- Out-of-order execution
  - Now we need to deal with WAR and WAW hazards besides RAW
  - Any of these hazards seem to ___________ re-ordering instructions and executing them out-of-order

Register Renaming

- WAR and WAW hazards can always be solved by simply choosing a ______________ register since no data is being ________________ but we were simply "reusing" a register
- If we had **64 registers** instead of **32 registers**, then perhaps the compiler might have used $48 instead of $8 and we could have executed the second part of the code before the first part

```
WAR = Write After Read
add $9, $8, $6
lw  $8____, 40($2)
WAW = Write After Write
add $9, $8, $6
lw  $9____, 40($2)
```

This is an example of a name-dependency

First iteration

```
lw   $8, 40($2)
add  $8, $8, $8
sw   $8, 40($2)
```

Second iteration (using alternate register, $48)

```
lw   $48, 60($3)
add  $48, $48, $48
sw   $48, 60($3)
```


Register Renaming

- Renaming requires more registers
- We have limited ______________ registers
  - Registers the instruction set is _________
- We could have more ___________ registers
  - Actual registers part of the register file

Assume Delayed

```
lw   $8, 40($2)
add  $8, $8, $8
sw   $8, 40($2)
```

```
lw   $8, 60($3)
add  $8, $8, $8
sw   $8, 60($3)
```

It is clear the compiler is using $8 as a temporary register

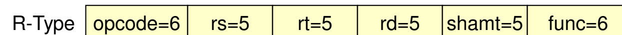
If there is a delay in obtaining $2 the first part of the code cannot proceed

Unfortunately, the second part of the code cannot proceed because of the name dependency for $8

Increasing Number of Registers

- Can a later implementation provide 64 registers (instead of 32) while maintaining ***binary compatibility*** with previously compiled code?
- Answer: Yes / No

- Why?
  Machine code has ______________________



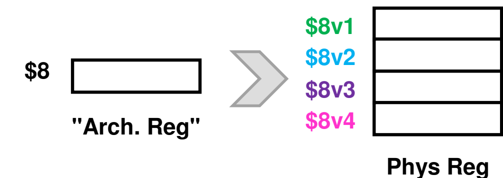
R-Type	opcode=6	rs=5	rt=5	rd=5	shamt=5	func=6

Register Renaming

- Rather than creating new architectural registers, let us internally provide multiple "versions" of the ____ _______________ register
  - $8v1 = $8 version 1
  - $8v2 = $8 version 2

```
lw   $8v1, 40($2)
add  $8v2, $8v1, $8v1
sw   $8v2, 40($2)
```

```
lw   $8v3, 60($3)
add  $8v4, $8v3, $8v3
sw   $8v4, 60($3)
```

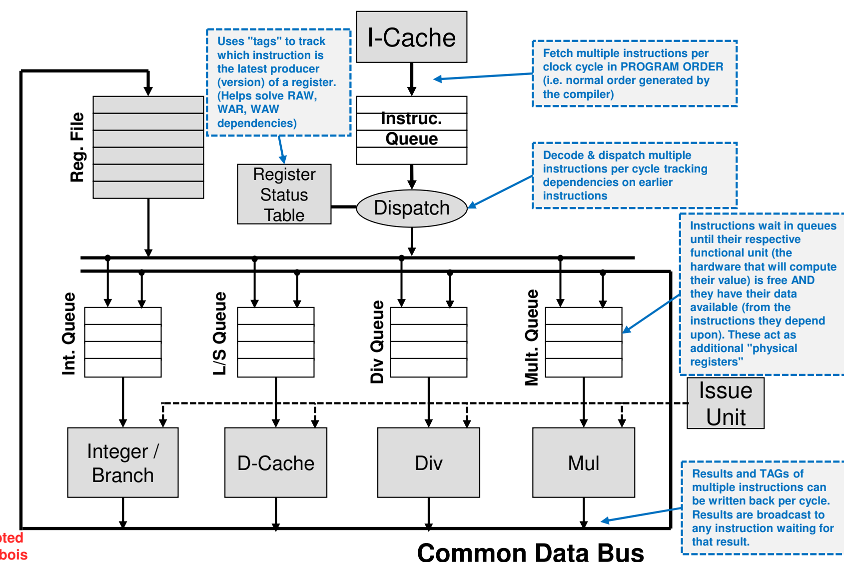


Tomasulo's Approach to Renaming

- Cannot change the number of architectural registers

- Instead we will perform
  Register Renaming through ***Tagging*** Registers
  - This solves name dependency problems (WAR and WAW) while attending to true dependency (RAW) through waiting in queues
  - Please be sure you understand this!

OoO Execution & Tomasulo's Algorithm



Block Diagram Adapted from Prof. Michel Dubois
(Simplified for EE457)

Tomasulo's Algorithm

- Dispatch/Issue unit decodes and dispatches instructions
- Assign a binary code (aka ______) to each instruction ______________ a register value using the TAG FIFO
- Adds a Register Status Table (RST) that holds the TAG of the instruction that is producing the ________________ version of each architectural register or NULL if the LATEST version is in the _________________
- The destination operand is represented by the TAG but not the actual register name
- For source operands, an instruction carries either the values (if TAG is null in RST) or TAGs of the operands (but not the actual register name)
- When an instruction executes and produces a result it broadcasts the result and its destination TAG
  - Any instruction waiting can compare its _________ tags with the _________ tag and grab the value if they match
  - If entry in RST matches the TAG then this instruction is the __________ producer of the register and the value will be written to the register file

Tagging process

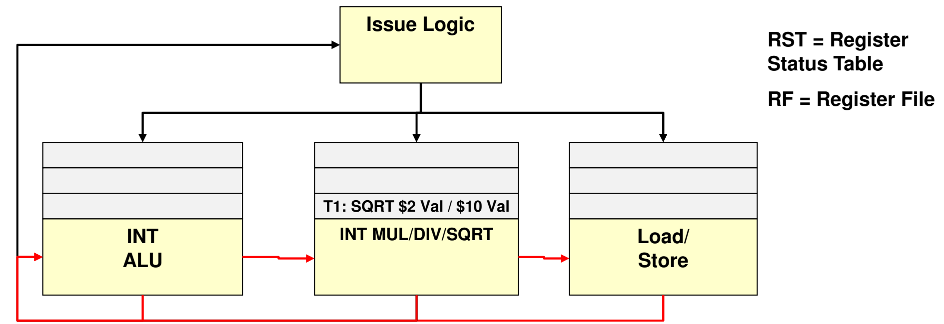
```
sqrt $2, $10
lw   $8, 40($2)
add  $8, $8, $8
sw   $8, 40($2)
lw   $8, 60($3)
add  $8, $8, $8
sw   $8, 60($3)
```

RST *(Identify latest version of a reg.)*

Reg	RST
$1	
$2	
$3	
$4	
$5	
$6	
$7	
$8	
…	
$31	

RF

Reg	RF
$1	
$2	
$3	
$4	
$5	
$6	
$7	
$8	
…	
$31	



Tagging process: CC1

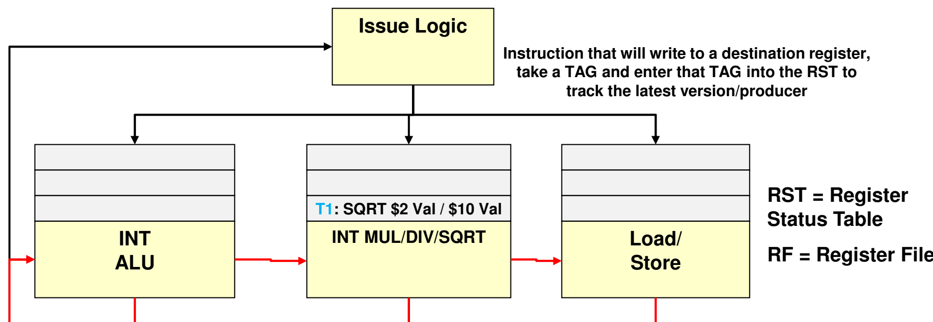
```
sqrt $2, $10
lw   $8, 40($2)
add  $8, $8, $8
sw   $8, 40($2)
lw   $8, 60($3)
add  $8, $8, $8
sw   $8, 60($3)
```

RST *(Identify latest version of a reg.)*

Reg	RST
$1	
$2	T1
$3	
$4	
$5	
$6	
$7	
$8	
…	
$31	

RF

Reg	RF
$1	
$2	
$3	
$4	
$5	
$6	
$7	
$8	
…	
$31	



Tagging process: CC3

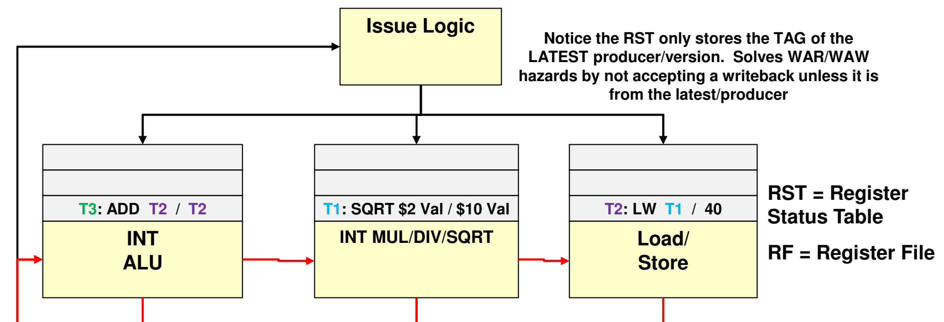
```
sqrt $2, $10
lw   $8, 40($2)
add  $8, $8, $8
sw   $8, 40($2)
lw   $8, 60($3)
add  $8, $8, $8
sw   $8, 60($3)
```

RST

Reg	RST
$1	
$2	T1
$3	
$4	
$5	
$6	
$7	
$8	~~T2~~ T3
…	
$31	

RF

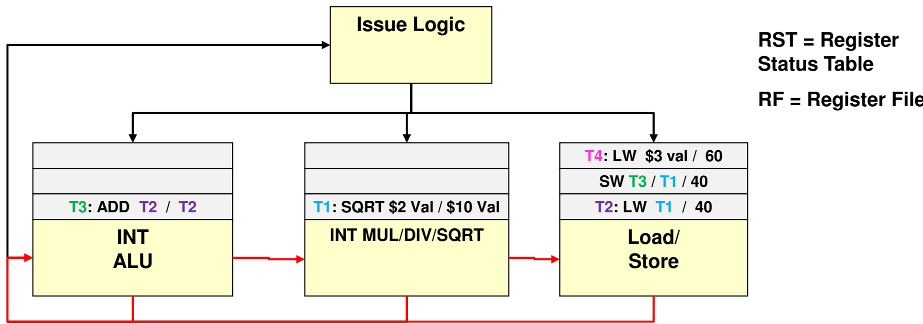
Reg	RF
$1	
$2	
$3	
$4	
$5	
$6	
$7	
$8	
…	
$31	



Tagging process: CC5

```
sqrt $2, $10
lw   $8, 40($2)
add  $8, $8, $8
sw   $8, 40($2)
lw   $8, 60($3)
add  $8, $8, $8
sw   $8, 60($3)
```

	RST	RF
$1		
$2	T1	
$3		
$4		
$5		
$6		
$7		
$8	~~T3~~ T4	
…		
$31		



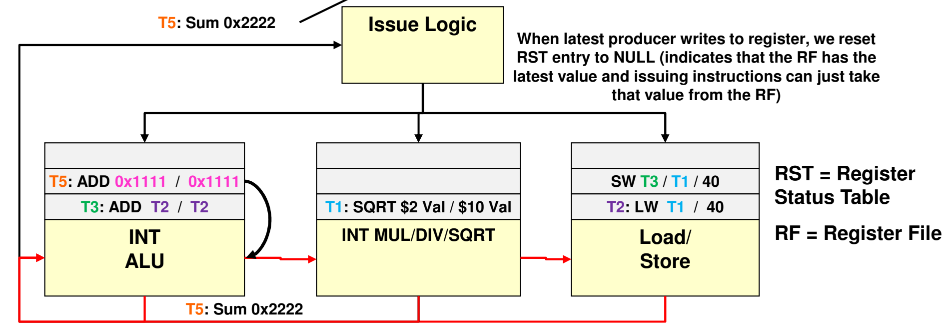
RST = Register Status Table

RF = Register File

Tagging process: CC8

```
sqrt $2, $10
lw   $8, 40($2)
add  $8, $8, $8
sw   $8, 40($2)
lw   $8, 60($3)
add  $8, $8, $8
sw   $8, 60($3)
```

	RST	RF
$1		
$2	T1	
$3		
$4		
$5		
$6		
$7		
$8	T5 => null	0x2222
…		
$31		



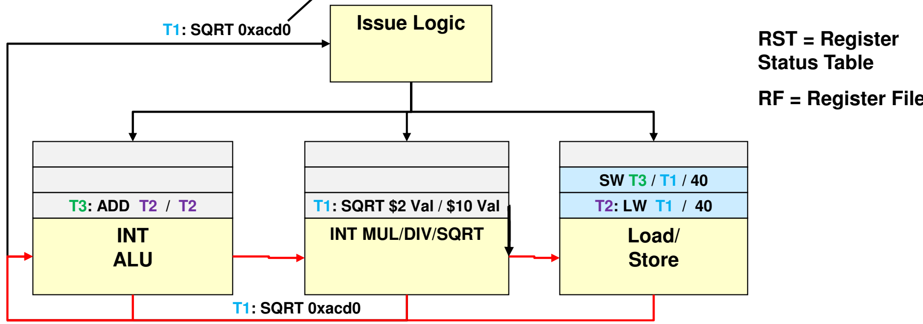
RST = Register Status Table

RF = Register File

Tagging process: CC10

```
sqrt $2, $10
lw   $8, 40($2)
add  $8, $8, $8
sw   $8, 40($2)
lw   $8, 60($3)
add  $8, $8, $8
sw   $8, 60($3)
```

	RST	RF
$1		
$2	T1 => null	
$3		
$4		
$5		
$6		
$7		
$8		0x2222
…		
$31		



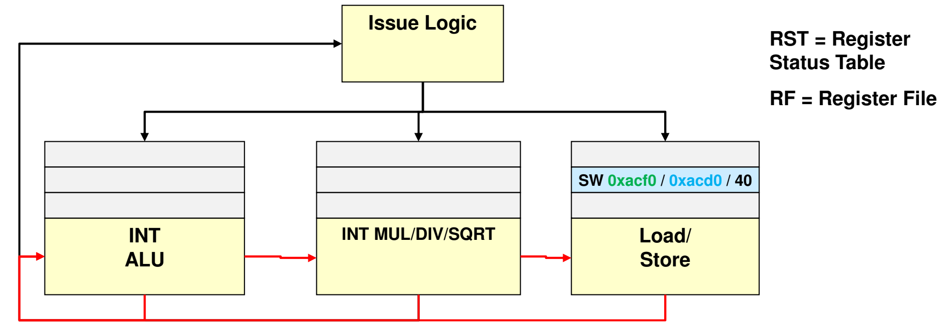
RST = Register Status Table

RF = Register File

Tagging process: CC13

```
sqrt $2, $10
lw   $8, 40($2)
add  $8, $8, $8
sw   $8, 40($2)
lw   $8, 60($3)
add  $8, $8, $8
sw   $8, 60($3)
```

	RST	RF
$1		
$2		0xacd0
$3		
$4		
$5		
$6		
$7		
$8		0x2222
…		
$31		

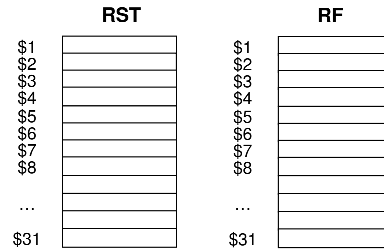


RST = Register Status Table

RF = Register File

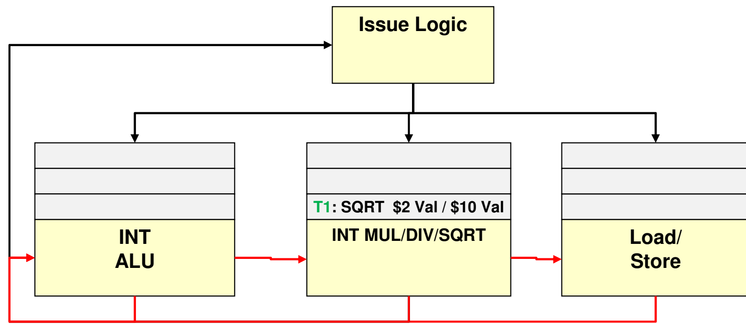
Register Renaming

```
sqrt $2, $10
add  $2, $2, $2
add  $2, $2, $2
add  $2, $2, $2
add  $2, $2, $2
```



RST = Register Status Table

RF = Register File



Unique TAGs

- Like SSN, we need a unique TAG
- SSN's are reused.
- Similarly TAGS can be reused
- TAGs are similar to number TOKEN

Helps to create a virtual queue.

We do not need that here

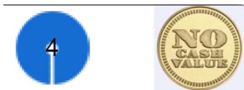
In State Bank of India, the cashier issues brass token to customers trying to draw money as an ID (and not at all to put them in any virtual queue / ordering). Token numbers are in random order.

The cashier verifies the signature in the record rooms, returns with money, calls the token number and issues the money.

Tokens are reclaimed & reused.

Tags (= Tokens)

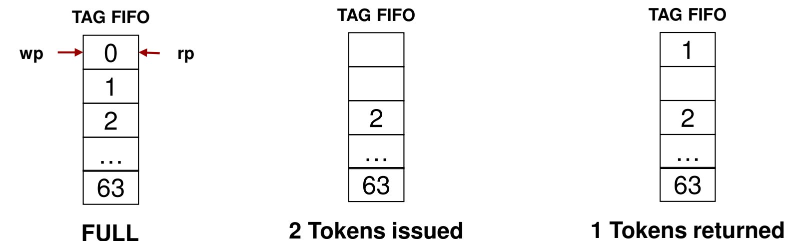
- How many tokens should the bank casheir have to start with?
- What happens if the tokens run out?
- Does the cashier need to have any order in holding tokens and issuing tokens?
- Do they have to collect the tokens back?



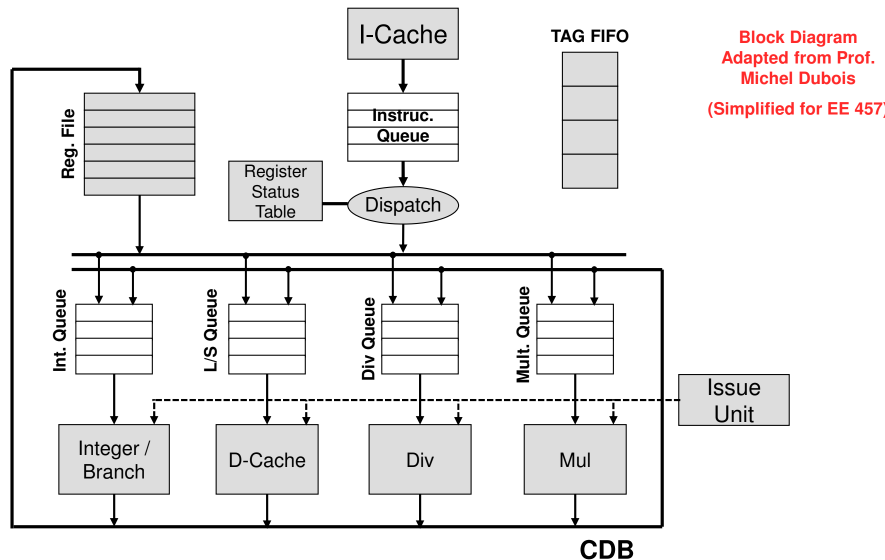
TAG FIFO

FIFO's are taught in EE 560

- To issue and collect tokens (TAGS) use a circular FIFO (First-In/First-Out) unit
  - While the FIFO order is not important here, a FIFO is the easiest to implement in hardware compared to a random order in a pile
- Filled (with say) 64 tokens (___________) initially on reset
- Tokens return ________________
- Put tokens back in the FIFO and ___________



Organization for OoO Execution



Front-End & Back-End

- IFQ (Instruction Fetch Queue)
  - A FIFO structure
- Dispatch (Issue) Unit
  - Includes RST, RF, Tag FIFO
- Load/Store and other Issue Queues
- Issue Units
- Functional units
- CDB (Common Data Bus)
  - Like a public address system that everyone can see/hear when data is produced

More Tomasulo Algorithm

- Front End
  - Instructions are fetched
  - They are stored in a FIFO (IFQ)
  - When instruction reached the head of the IFQ it is
    - Decoded
    - Dispatched to an issue queue/functional unit
    - Even if some of the inputs are not ready (takes TAGs)
- Back End
  - Instructions in issue queues wait for their input operands
  - Once register operands are ready instructions can be scheduled for execution provided they will not conflict for the CDB or their functional unit
  - Instructions execute in their functional unit and their result is put on the CDB
  - All instructions in queues and the register file "watch" the CDB and grab the value they are waiting for when it is produced
- Bottleneck in Tomasulo's algorithm?
  - The _________!
  - Do all instructions use the __________? __________________________________

Data hazards and memory

MEMORY DISAMBIGUATION

Load/Store Queue (LSQ)

- For our course, the **LSQ** performs
  - Address calculation
  - Memory disambiguation
    - _________________________ hazards due to memory reads and writes

```
// Is there a dependency here?
SW $2,0($5)
LW  $8,0($5)
// What about here?
SW $2, 1000($4)
LW $3, 0($6)
```


Memory Disambiguation

- Data hazards (RAW, WAR, WAW) can occur in memory just as with registers, and hazards in memory are much harder to deal with since many ___________ could produce the same address

```
RAW
sw $2, 2000($0)
lw $8, 2000($0)
```

This later lw can proceed only if there is no store ahead of it with the same address

```
WAW
sw $2, 2000($0)
sw $8, 2000($0)
```

This later sw can proceed only if there is no store ahead of it with the same address

```
WAR
lw $2, 2000($0)
sw $8, 2000($0)
```

This later sw can proceed only if there is no load ahead of it with the same address

Address Calculation for LW/SW

- EE 557 approach for address calculation
  - Loads & store in 2 sub-instructions
    - 1 instruction computes address and is dispatched to integer ALU
    - 1 instruction access data cache and is issued to LSQ
    - Address is communicated from integer ALU to LSQ via CDB forwarding using a tag
- EE 560/457 approach
  - Use a dedicated adder in the LSQ to compute address (so just 1 dispatched instruction)

Memory Disambiguation

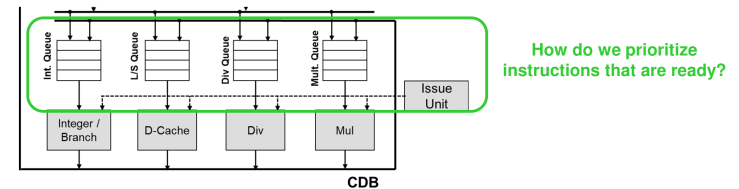
- When can LSQ can issue a LW or SW to cache?
  - Loads can issue to a cache when their address is ready
  - Stores can issue to cache when both address & data is ready
  - Memory hazards (RAW, WAR, WAW) are resolved in the LSQ
    - Load can issue to cache if no _________ with same address is before it
    - Store can issue to cache if no _________ or _________ with same address before it
    - Otherwise, access waits in LSQ
  - If an address is unknown it is assumed to be the ___________
    - Worst case to enforce correctness
  - The process of figuring out and comparing memory address is called “disambiguation”

Issue Queue priority, Branches, etc.

LAST CONSIDERATIONS FOR OUT-OF-ORDER EXECUTION/COMPLETION

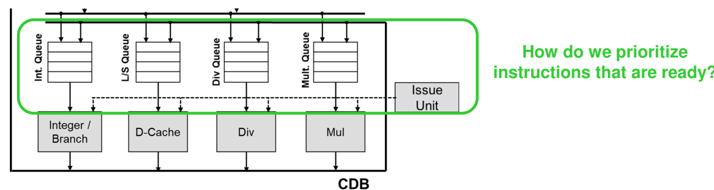
Issue Unit

- How do we determine when to issue an instruction to the functional unit?
  - Is the instruction ready
  - Is the functional unit free to start the operation?
  - CDB availability constraint
    - Will there ___________________ when operation finished?
  - Priority/conflict resolution
    - If many instructions are available, which should be chosen? (Is round-robin priority adequate)?



Issue Queue Priority

- Priority (based on the order of arrival among ready instructions)
  - Is it necessary or just desirable?
  - Local priority within queues?
  - Global priority across the queues?



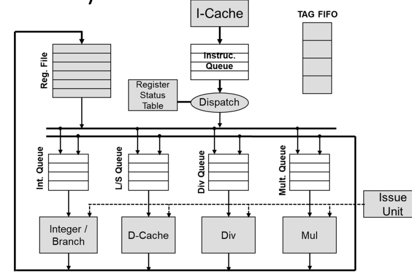
LSQ Ordering/Priority

- Maintaining instructions in the order of arrival
  - Issue order/program order in a queue
- Is this necessary and/or desirable?
  - In the case of LSQ?
    - ______________________________________
  - In the case of Integer, MUL, DIV queues?
    - Desirable, so that an earlier instruction gets executed whenever possible, thereby reducing queue pressure from too many instructions waiting on it

Conditional Branches

- Dispatcher stalls when it reaches a branch (and waits until it is resolved)
- Branches are dispatched to integer queue where they wait for their operands (if necessary)
- When branch executes it puts its outcome & target on CDB
  - If untaken, dispatch unit resumes
  - If taken, then dispatch clears flushes the IFQ and resumes at target
- Since we stop dispatching instructions after a branch, does it mean that **this branch is the last instruction** to be executed in the back-end?
- Is it possible that the back-end holds simultaneously
  - A. Some instructions dispatched before the branch .. AND ..
  - B. Some instructions issued after the branch

```
     ADD $4,$5,$5
     BEQ $6,$7,L1
     ...
L1:  SUB $1,$2,$3
     MUL $9,$7,$2
```



Structural Hazards + Exceptions

- Structural Stalls
  - Dispatch must stall if ___________ OR all entries in the **desired** functional unit's issue queue are occupied AND an instruction of that type is attempting to dispatch
  - Fetch unit must stall if the _________
  - Functional units stall when no ready instructions in the queue or CDB scheduling conflicts
- Precise exceptions not supported
  - Some instructions ________ the offending instruction may have updated registers or memory! _____!
  - We'll handle this in the next unit