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)

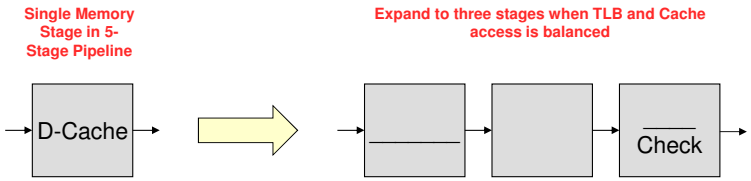


Outline

- _____ Parallelism
  - In-order pipeline
    - From academic 5-stage pipeline
    - To 8-stage MIPS R4000 pipeline
  - Superscalar, superpipelined
  - Out-of-Order Execution
    - Out-of-order completion (Problem: Exceptions)
    - In-order completion
- _____ Parallelism
  - Chip _____ (CMT)
  - Chip _____ (CMP)

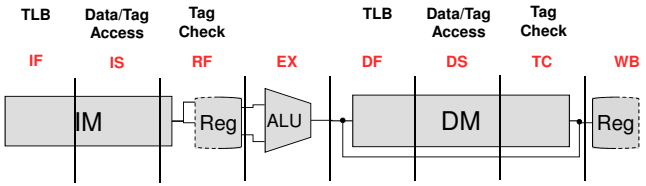
8-Stage Pipeline

- In 5-stage pipeline, we place the I-Cache and D-Cache in one stage each
- When we add VM translation (i.e. TLB), cache tag check, and data access we need more stages to balance the delay



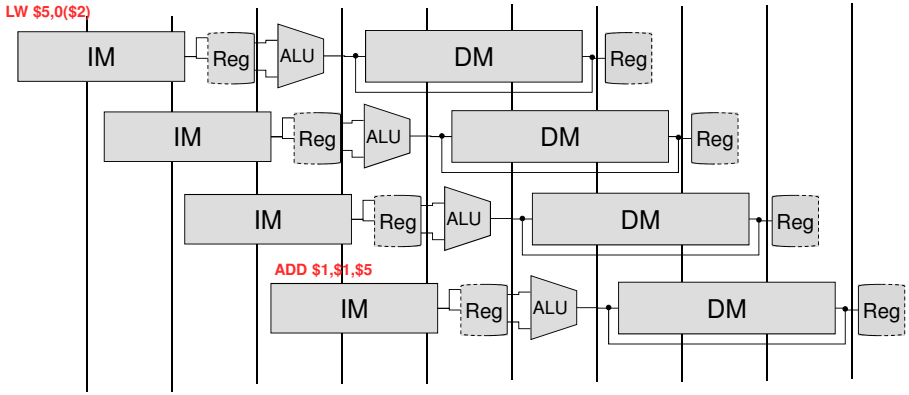
8-Stage Pipeline

- R4000 uses pipelined instruction and data caches
- The instruction is available at the end of the IS stage but the tag check is done in RF
  - This is harmless since it's fine to start the decode and *read* registers even if it's an invalid tag so long as we know by the end of the clock
- For the data cache we must perform the tag check before we can start writing into the register file



Load Dependency Issue Latency

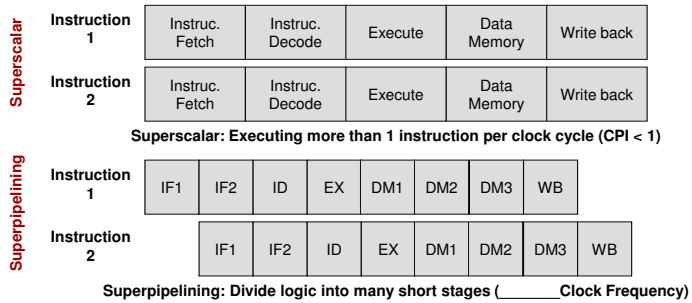
- 8-stage pipeline has a _____
  - Possible since data is available at end of DS and can be forwarded
  - If tag check indicates a miss the pipeline can be "backed up" (re-execute the instruction when the data is available



SUPERSCALAR & SUPERPIPELINING

Overview

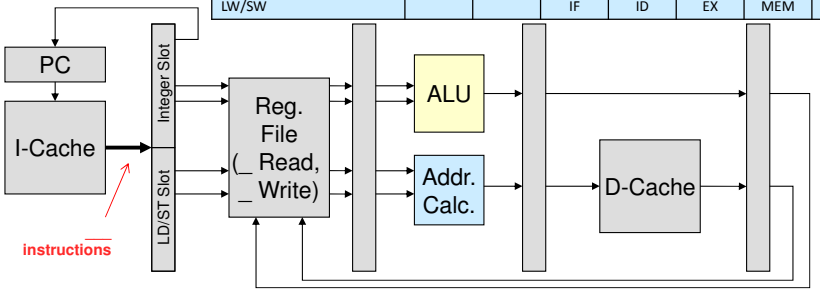
- Superscalar = More than 1 instruction _____
  - _____ superscalar = Proc. that can issue 2 instructions per clock cycle
  - Success is sensitive to ability to find independent instructions to issue in the same cycle
- Superpipelining = Many small stages to boost _____.
  - Success depends of finding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



2-way Superscalar

- One ALU & Data transfer (LW/SW) instruction can be issued at the same time

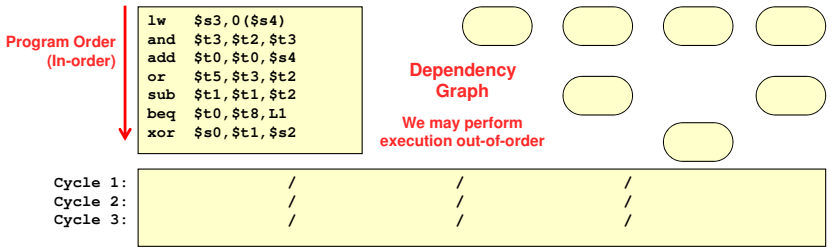
Instruction	Pipeline Stages						
ALU or branch	IF	ID	EX	MEM	WB		
LW/SW	IF	ID	EX	MEM	WB		
ALU or branch		IF	ID	EX	MEM	WB	
LW/SW		IF	ID	EX	MEM	WB	
ALU or branch			IF	ID	EX	MEM	WB
LW/SW			IF	ID	EX	MEM	WB



OUT-OF-ORDER EXECUTION

Instruction Level Parallelism (ILP)

- Although a program defines a sequential ordering of instructions, in reality many instructions can be executed in parallel.
- ILP refers to the process of finding instructions from a single program/thread of execution that can be executed in parallel
- _____ is what truly limits ordering
- _____ instructions (no data dependencies) can be executed at the same time)
- _____ also provide some ordering constraints



Out-of-Order Motivation

- Hide the impact of dynamic events such as a cache miss
- Out-of-Order (OoO) Execution
  - Let independent instructions behind a stalled instruction execute
  - Important aspect: Completion Ordering
    - Out-of-Order completion: Let the independent instruction that has been executed _____ before the stalled instruction completes
      - Problem: Exception handling
    - In-Order completion: Let the independent instructions execute but _____ until the stalled instruction completes

Out-of-Order Execution

- Io = In-order Execution
- "Execution" here means producing the results not necessarily writing them to a register or memory
- _____ means committing/writing the results to register file or memory
- While we say out-of-order execution we really mean:
  - IoI (IoD) => OoOE => IoC
  - IoI / IoD = In-order Issue/Dispatch
  - IoC = In-order completion

IoC or OoC?

- IoI (IoD) => OoOE => IoC
  - In-order completion is necessary to support _____ [_____]
- We will present the concept of OoOC (out-of-order completion) then roll back to IoC
- OoOC Issues
  - Branches…we should not commit an instruction that came after (in program order) a branch
  - Solution: _____ instructions after a branch until we resolve the _____

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


Basic Blocks

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

```
     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

Scheduling Strategies

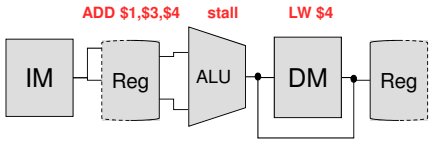
- _____ Scheduling
  - _____ re-orders instructions in such a way that no dependencies will be violated and allows for OoOE
- _____ Scheduling
  - ____ implementing the _____ algorithm or other similar approach will re-order instructions to allow for OoOE
- More Advanced Concepts
  - Branch prediction 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

Where to Stall?

- In 5-stage pipeline (in-order execution) RAW dependency was solved by
  - _____
  - _____
- Dependent instructions stalled in the ID stage if necessary

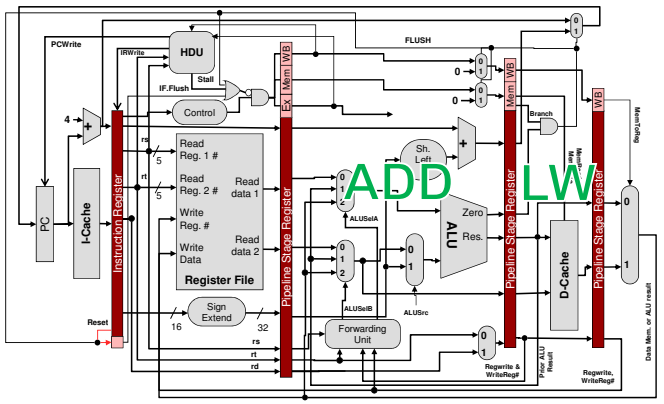


Where to Stall?

- Simple 5-stage pipeline:
  - Dependent instructions cannot be stalled in the EX stage

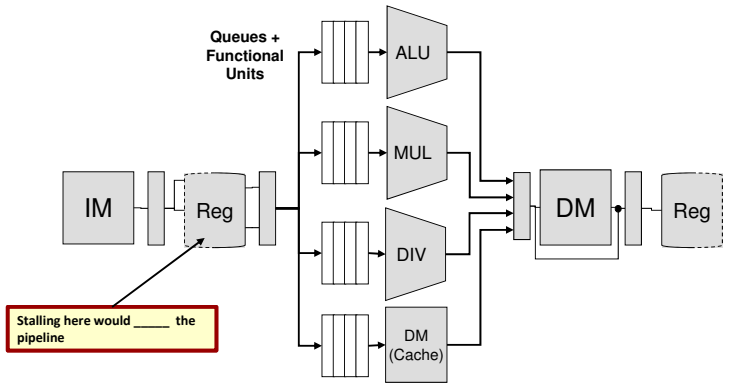
Why? What if ADD was also dependent on the _____

Thus we stall in ID so we can use the _____ to grab dependent values. Further stalling in ID incurs only 1 cycle penalty as would stalling in EX.



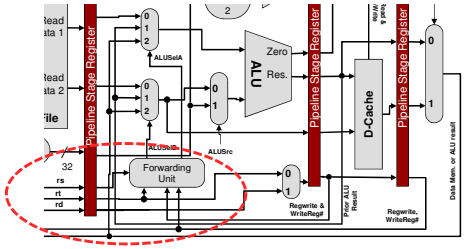
Where to Stall?

- But to implement OoO execution, we _____ 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 junior instructions carried their source register IDs into the EX stage to be compared with _____ register ID's of their _____ (_____ instructions)
- But in OoO execution, we may have many instructions (_____) in front of us and cannot afford to perform so many comparisons (as well as handling the case where many seniors are producing _____ of a register)
- Instead, the dispatch unit will _____ the dependent instruction _____



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

OoO Execution Problems

- For the time
  - No branch prediction
  - No speculative execution beyond a branch
- So we simply stall on a conditional branch
- For the time, no support for precise exceptions
  - Even then…

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
  - lw $8, 40($2)
- WAW = Write After Write
  - add $9, $8, $6 ← say $6 is not available yet
  - 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 it possible?
  - In OoO execution, instructions before the branch and after the branch can co-exist
  - Consider multiple iterations of a loop body

```
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
```


Another WAW Example

- 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

```
int x = _________________;
if (_____________)
  _____________________;
set_bonus(x);
```


RAW, WAR, and WAW

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

RAW
A true dependency

- 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

WAR
An _____

WAW
An _____

Note: no information is communicated in WAR/WAW 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 prevent re-ordering instructions and executing them out-of-order

Register Renaming

- We have limited _____ registers
  - Registers the instruction set is aware of
- 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

Register Renaming

- If we had __ **registers** instead of __ **registers**, then perhaps the compiler might have used $__ instead of $8 and we could have executed the second part of the code before the first part

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

```
lw   ___, 60($3)
add  $__, $__, $__
sw   $__, 60($3)
```

This is an example of a name-dependency

Register Renaming

- Four different temporary registers can be used here as shown: $8, $18, $28 and $48
- We could even simply use aliases or code names: LION, TIGER, CAT, BEAR

```
lw   $8, 40($2)
add  $18, $8, $8
sw   $18, 40($2)
```

```
lw   $28, 60($3)
add  $48, $28, $28
sw   $48, 60($3)
```

```
lw   LION, 40($2)
add  TIGER, LION, LION
sw   TIGER, 40($2)
```

```
lw   CAT, 60($3)
add  BEAR, CAT, CAT
sw   BEAR, 60($3)
```


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?

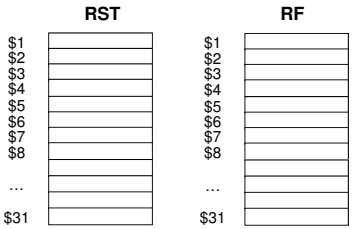
Increasing Number of Registers

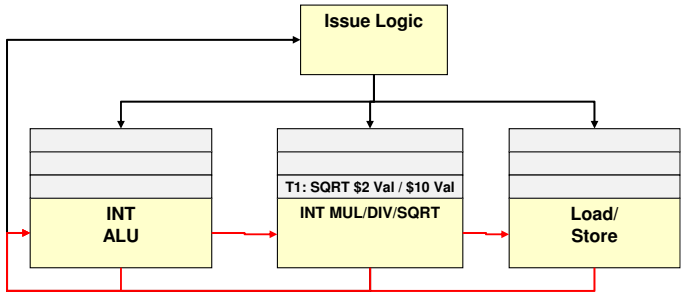
- **Answer**: _____ the number of _____ registers

- Instead we will perform Register Renaming through _____ 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!

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)
```



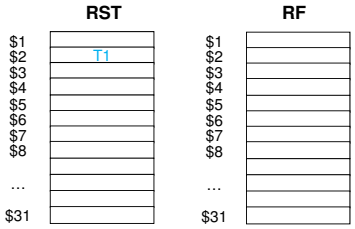


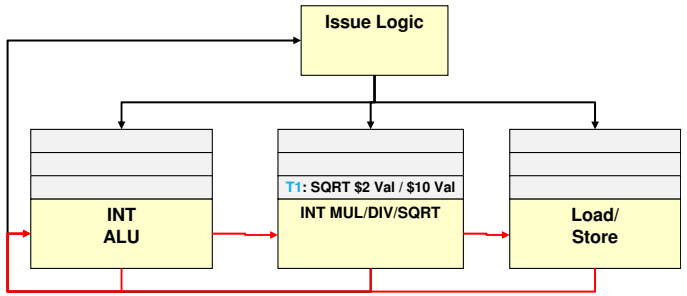
RST = Register Status Table

RF = Register File

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)
```



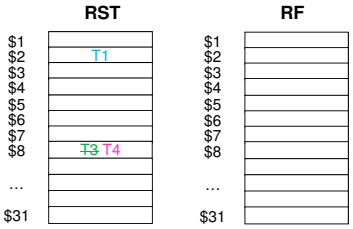


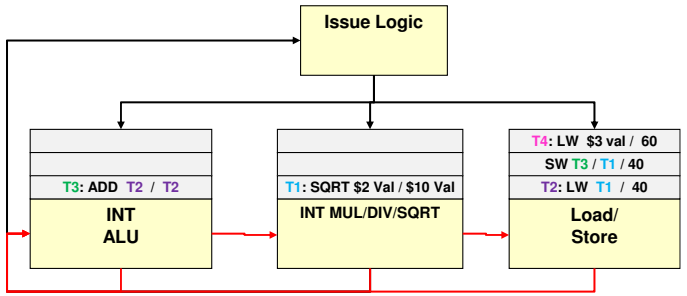
RST = Register Status Table

RF = Register File

Tagging process: CC__

```
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)
```

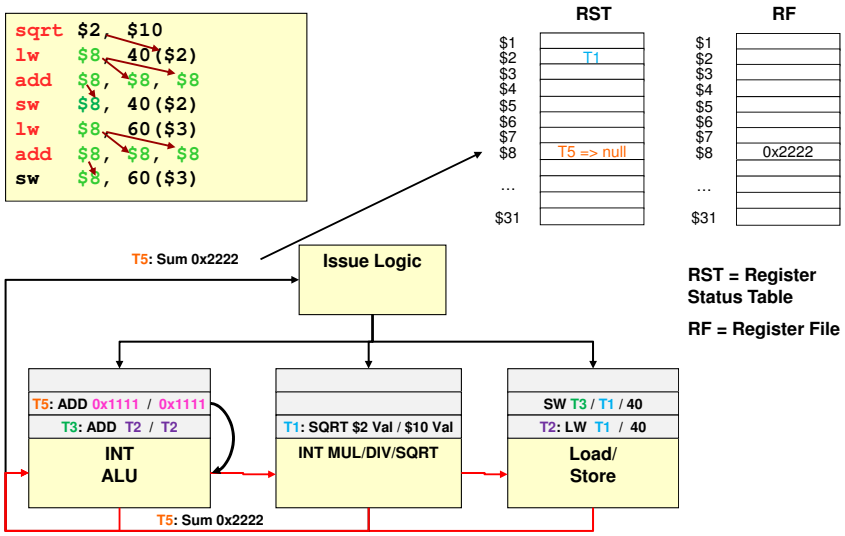




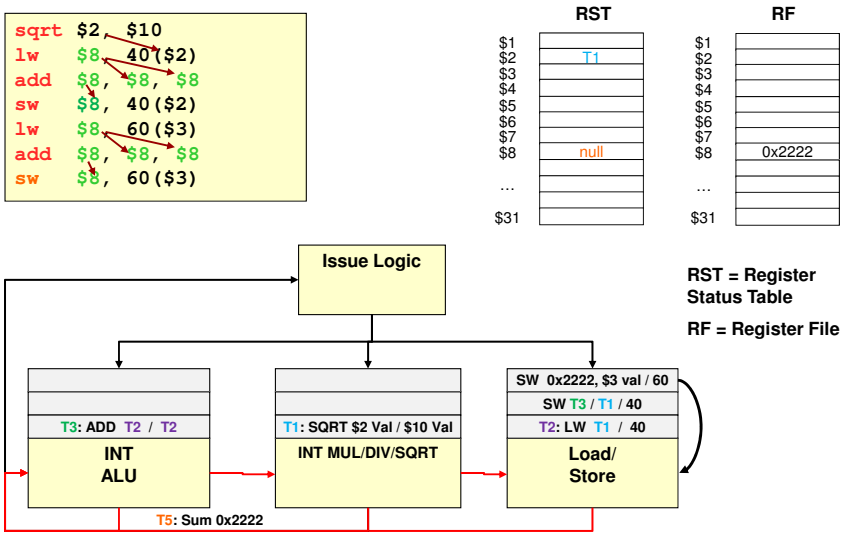
RST = Register Status Table

RF = Register File

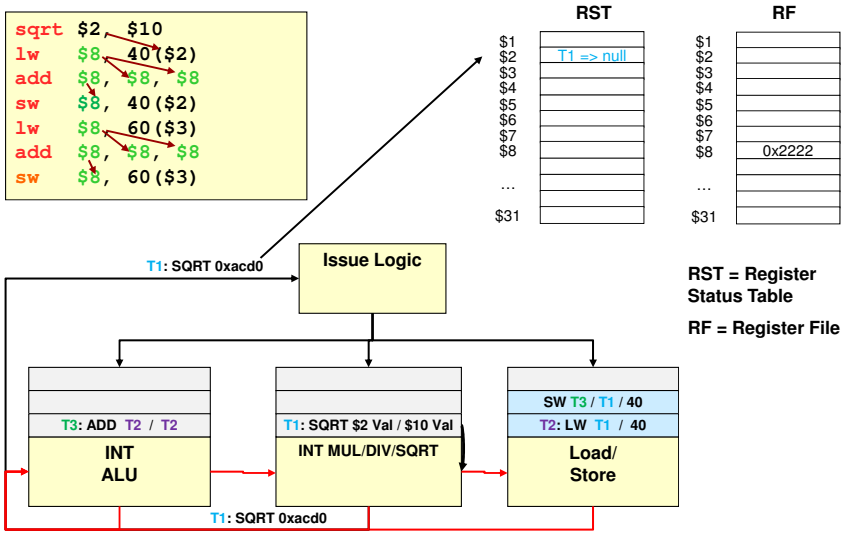
Tagging process: CC8



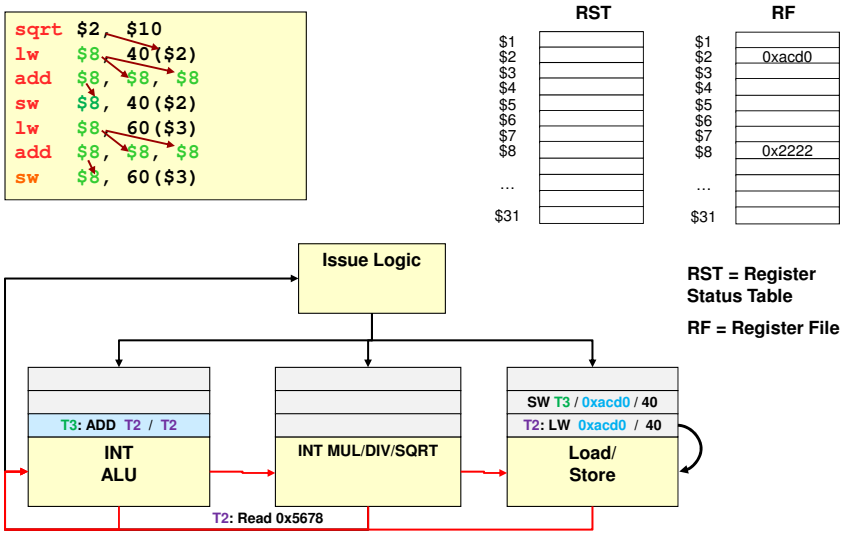
Tagging process: CC9



Tagging process: CC10



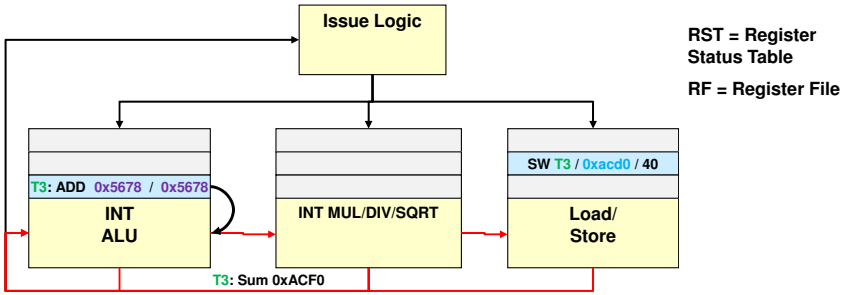
Tagging process: CC11



Tagging process: CC12

```
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		



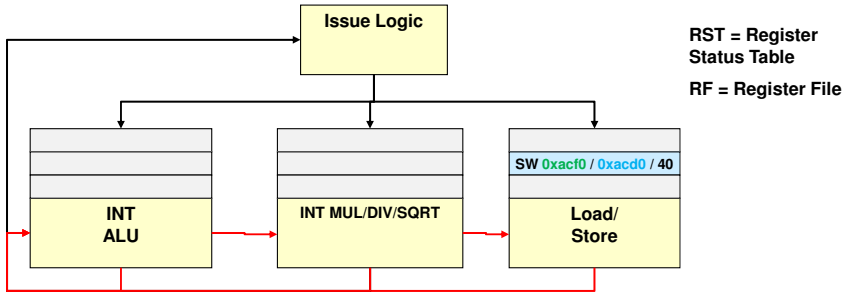
RST = Register Status Table

RF = Register File

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

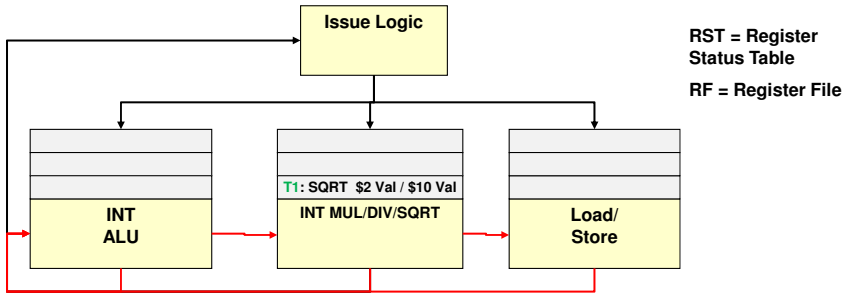
Tomasulo's Algorithm

- Dispatch/Issue unit decodes and dispatches instructions
- For destination operand, an instruction carries a ____ (but not the actual register name)
- For source operands, an instruction carries either the _____ (if TAG is _____) or _____ 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 _____ with the _____ tag and _____ 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 RF

Register Renaming

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

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



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.

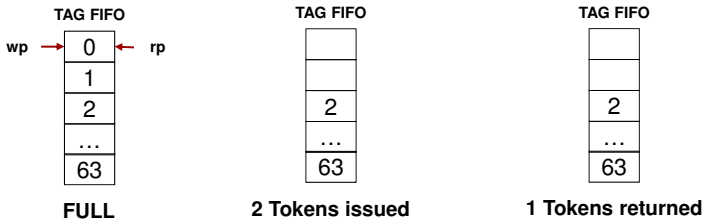
Tags (= Tokens)

- 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?

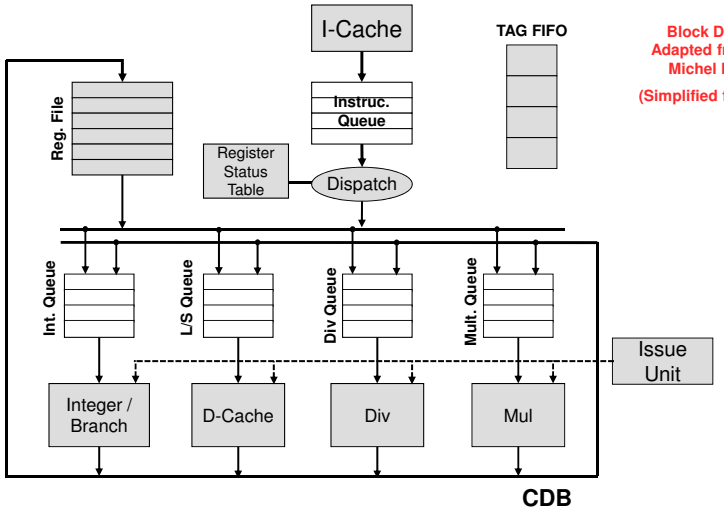
TAG FIFO

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



Block Diagram Adapted from Prof. Michel Dubois

(Simplified for EE 457)

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 Approach

- _____
- Do all instructions use the _____?
  -

Load/Store Queue (_____)

- 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)
```

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

- EE 557: Issue to a cache from LSQ
  - 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 store with same address is before it
    - Store can issue to cache if no store or load with same address before it
    - Otherwise access waits in LSQ
  - If an address is unknown it is assumed to be the same
    - Worst case to enforce correctness
  - The process of figuring out and comparing memory address is called "disambiguation"

Memory Disambiguation

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

LSQ Ordering

- 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?
    - _____, so that an earlier instruction gets executed whenever possible, thereby reducing queue pressure from too many instructions waiting on it

Priority

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

Issue Unit

- CDB availability constraint
- Pipelined functional unit vs. multicycle unit
- Conflict resolution
  - Round-robin priority adequate?, well, …

Conditional Branches

- Dispatch unit stops dispatching until the branch is resolved
- CDB broadcasts the result of the branch
- Dispatching continues at the target or fall-through instruction
- Successful branch shall cause flushing of _____
- 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

Structural & Control Hazards

- Structural Stalls
  - I-Fetch must stall if the _____
  - Dispatch must stall if all entries in the desired _____ _____ is occupied
  - Instructions cannot be scheduled in case of _____ or functional unit
- Control
  - Dispatcher stalls when it reaches a branch
  - Branches are dispatched to integer queue
  - They wait for their operands if needed
  - Put their result on CDB
    - If untaken, dispatch unit resumes
    - If taken, then dispatch clears the IFQ and resumes at target
- Precise exceptions not supported