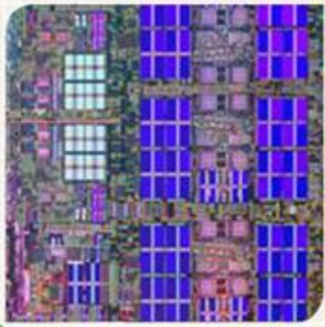
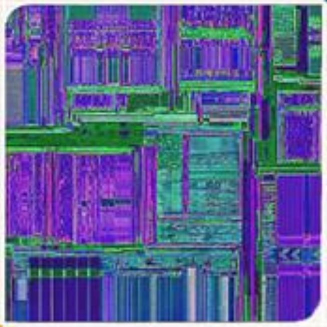



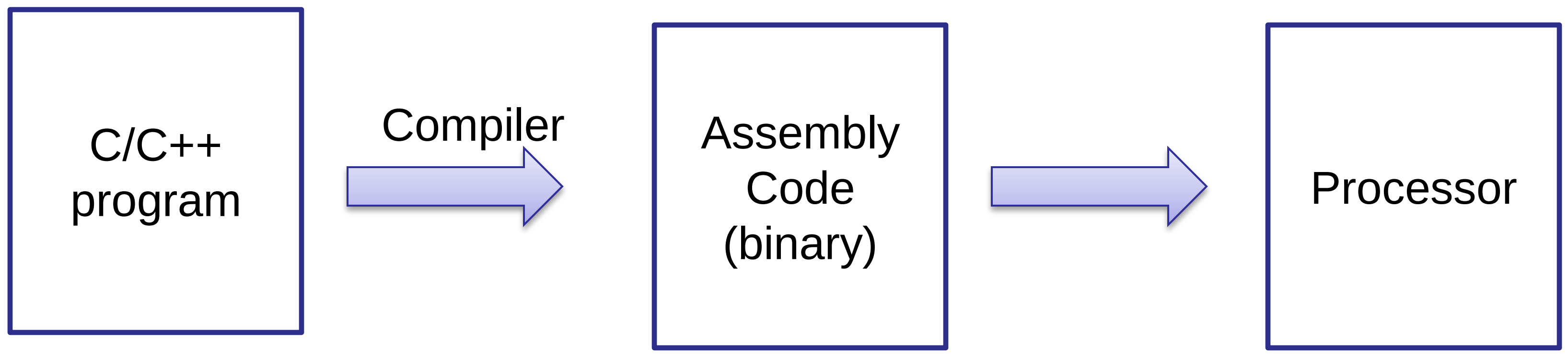
CS4803DGC Design Game Consoles

Spring 2010

Prof. Hyesoon Kim

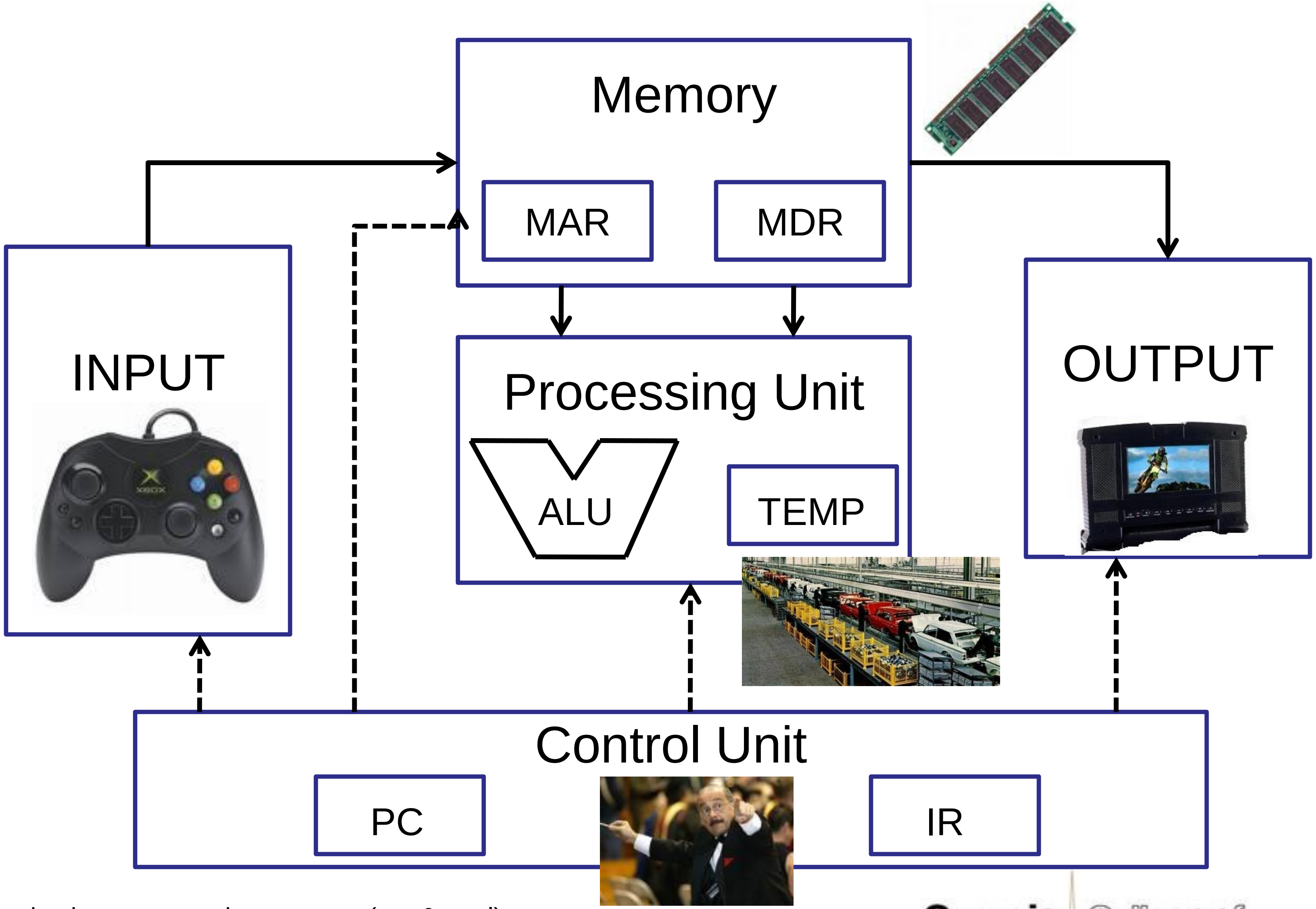
Georgia Tech College of Computing

Thanks to Prof. Loh & Prof. Prvulovic



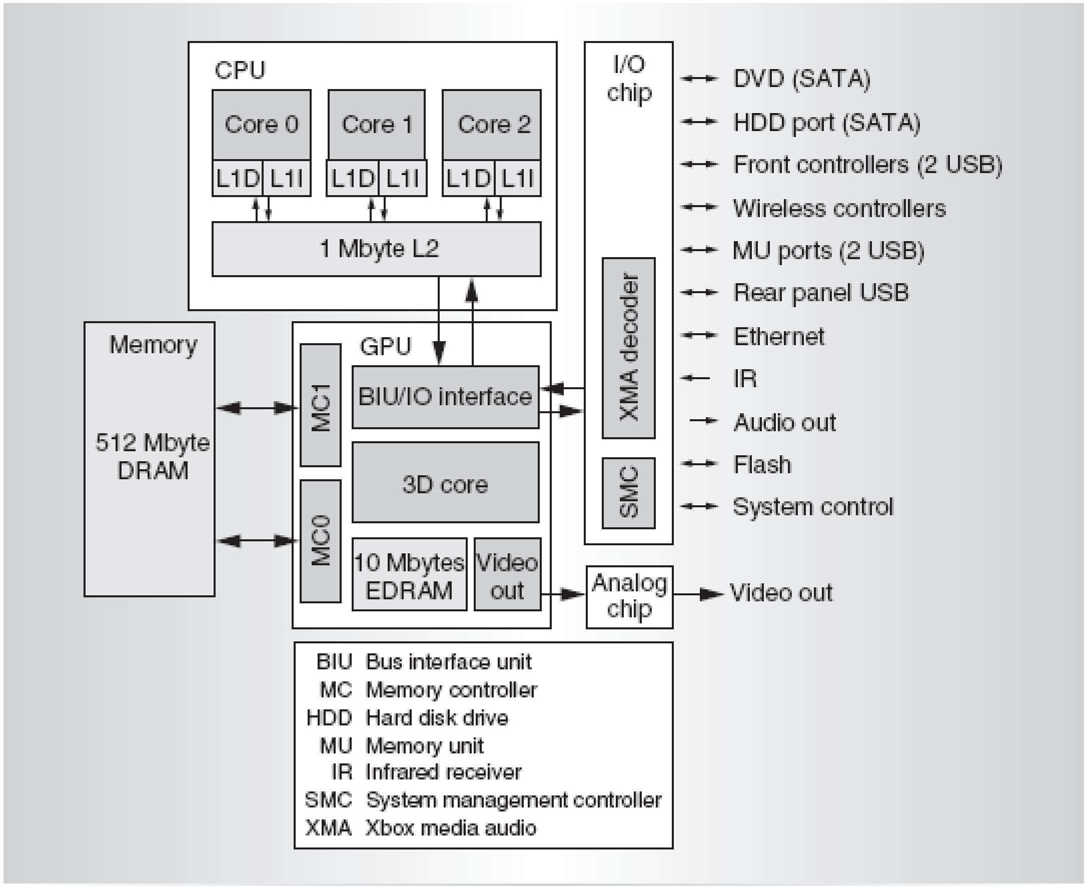
0010101010101011110

Von Neumann Model



- http://www.youtube.com/watch?v=_Lm7Acr5ysY&feature=related

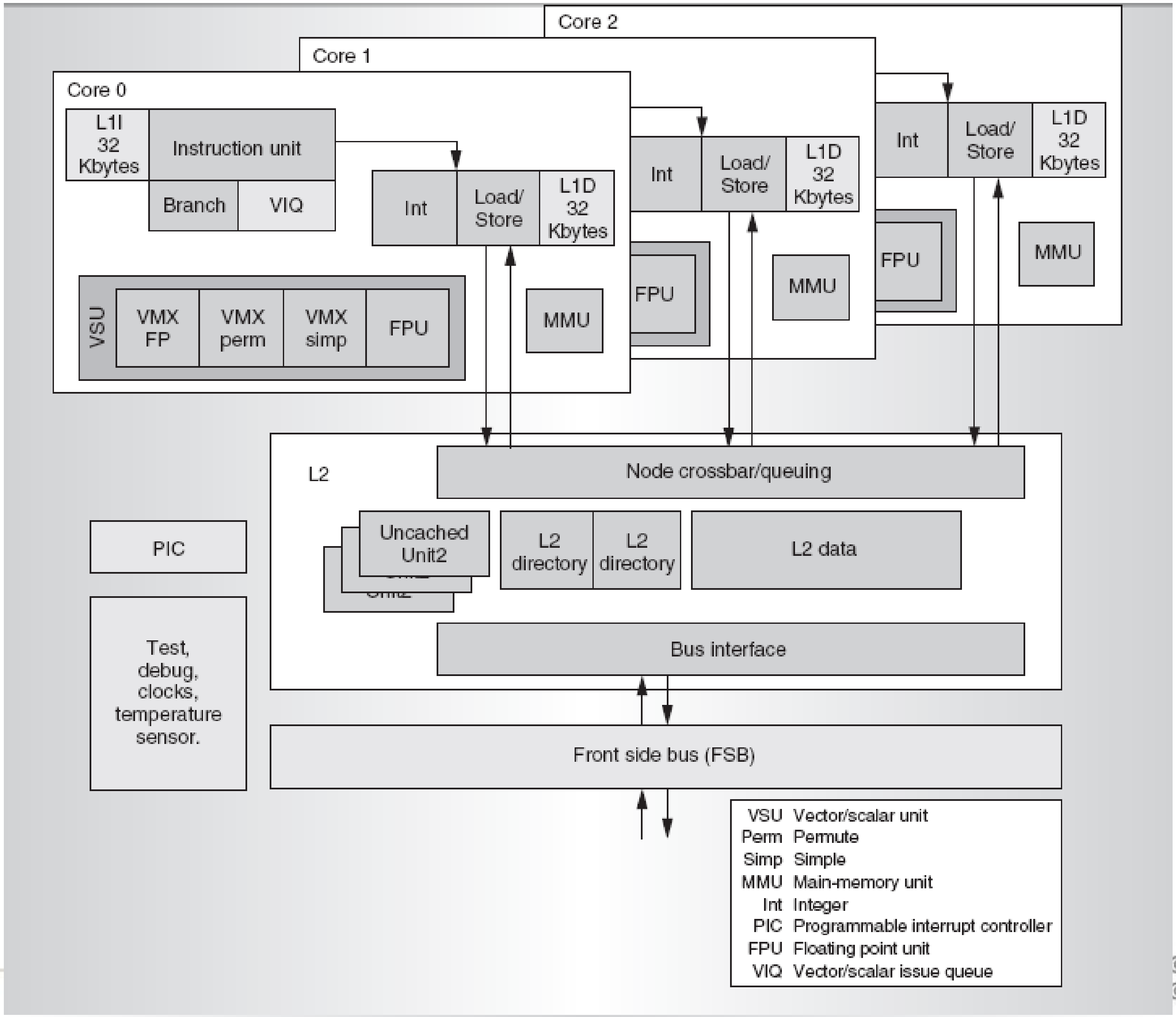
Xbox 360 System Block Diagram



BIU	Bus interface unit
MC	Memory controller
HDD	Hard disk drive
MU	Memory unit
IR	Infrared receiver
SMC	System management controller
XMA	Xbox media audio

Figure 2. Xbox 360 system block diagram.

Xbox 360 CPU Block Diagram

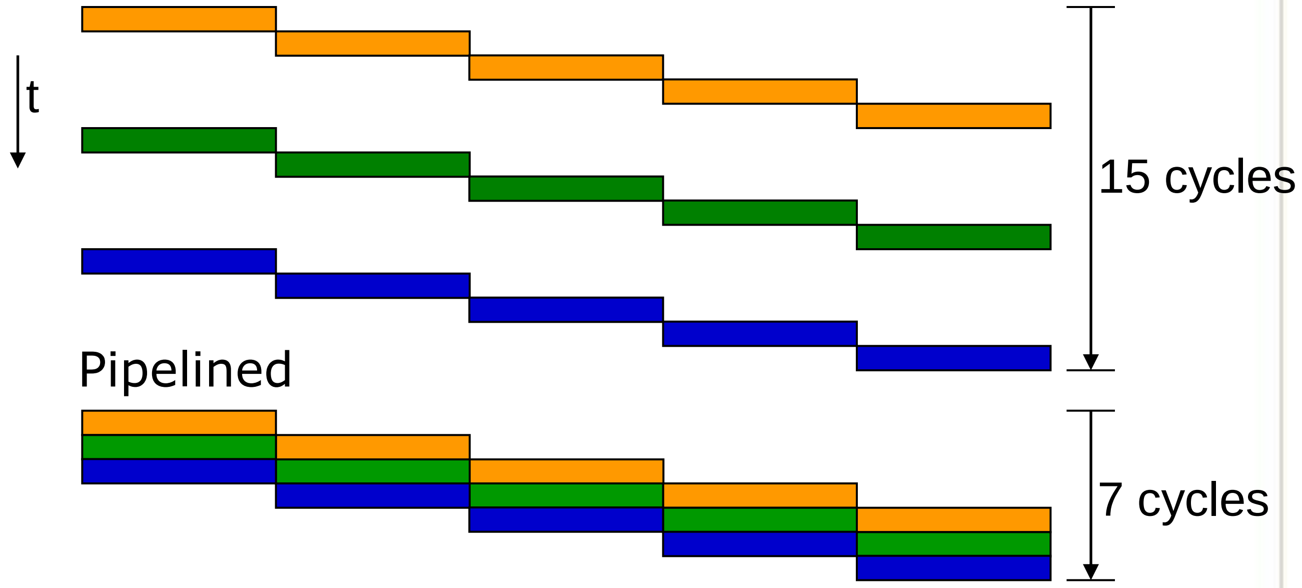


VSU	Vector/scalar unit
Perm	Permute
Simp	Simple
MMU	Main-memory unit
Int	Integer
PIC	Programmable interrupt controller
FPU	Floating point unit
VIQ	Vector/scalar issue queue

PROCESSOR DESIGN

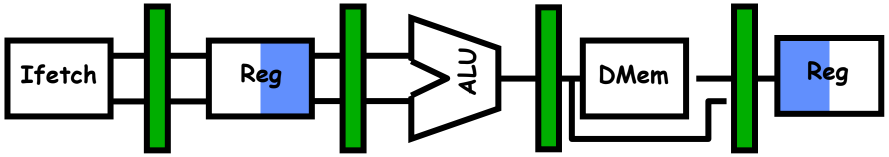
Overview of a Processor

0x0001	LD R1, MEM[R0]
0x0002	ADD R2, R2, #1
0x0003	BRZERO 0x0001



Dependences/Dependencies

- Data Dependencies
  - RAW: Read-After-Write (True Dependence)
  - WAR: Anti-Depedence
  - WAW: Output Dependence
- Control Dependence
  - When following instructions depend on the outcome of a previous branch/jump



Dynamic scheduling

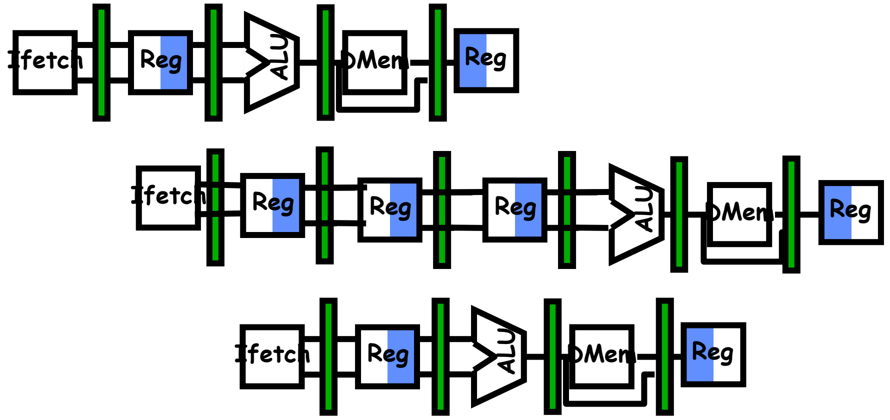
Instr. Order

```
add r1,r2,r3

sub r4,r1,r3

and r6,r2,r7
```

All sources are ready?
Why not execute them?



Impact of Ignoring Dependencies

Read-After-Write

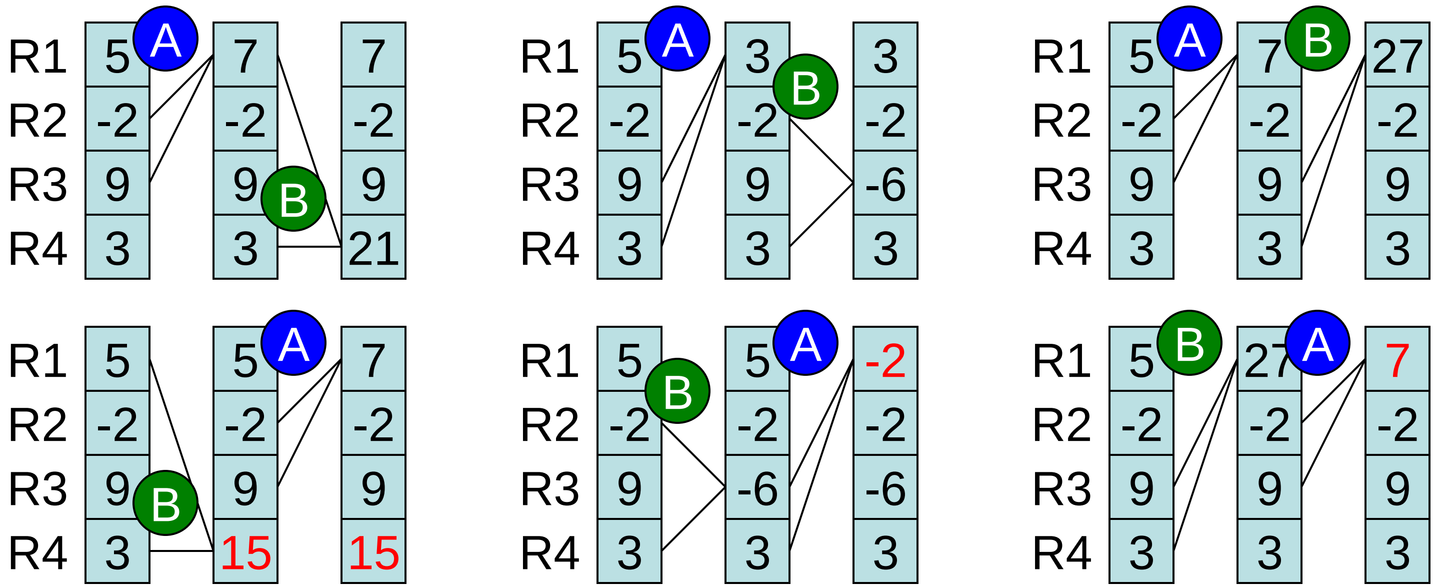
A: R1 = R2 + R3
B: R4 = R1 * R4

Write-After-Read

A: R1 = R3 / R4
B: R3 = R2 * R4

Write-After-Write

A: R1 = R2 + R3
B: R1 = R3 * R4

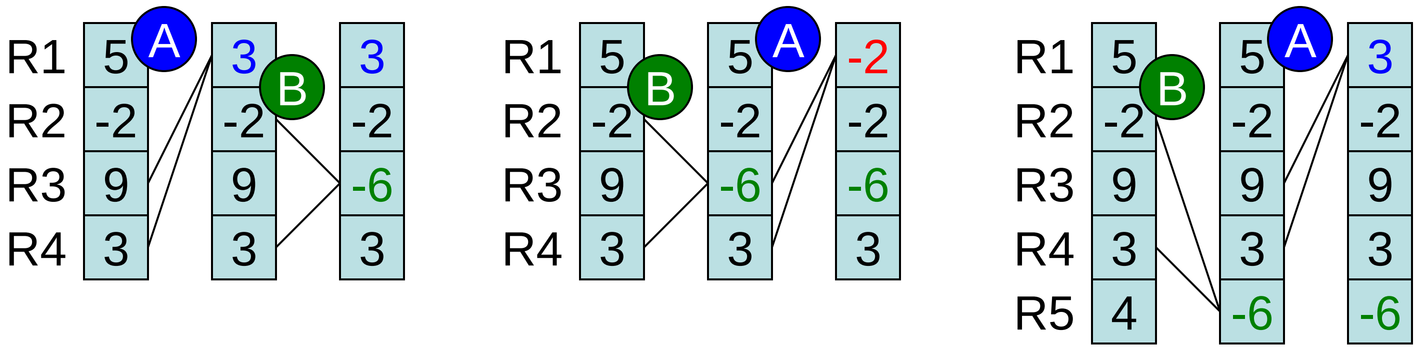


Eliminating WAR Dependencies

- WAR dependencies are from reusing registers

A: R1 = R3 / R4
B: R3 = R2 * R4

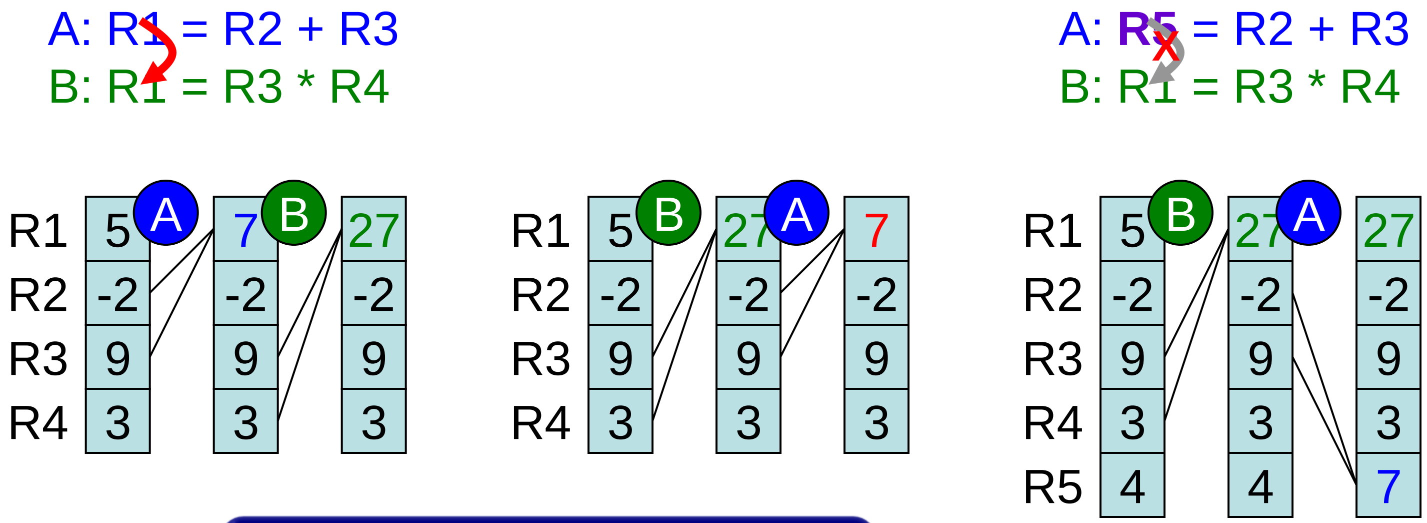
A: R1 = R3 / R4
B: **R5** = R2 * R4



With no dependencies, reordering still produces the correct results

Eliminating WAW Dependencies

- WAW dependencies are also from reusing registers



Same solution works

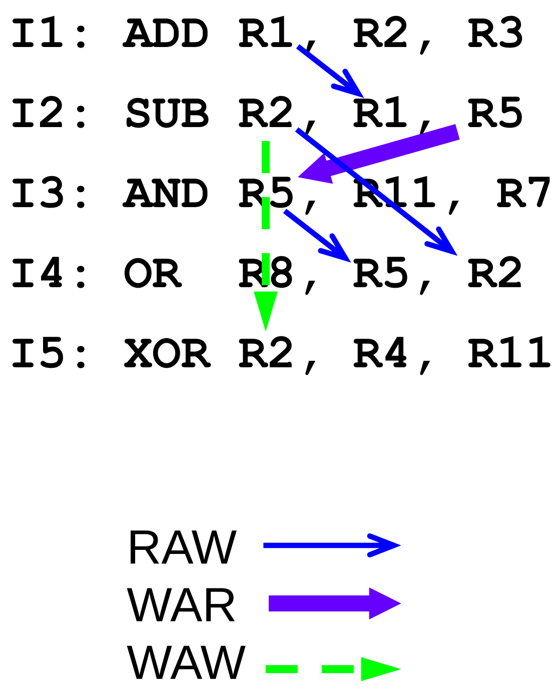
Better Solution: HW Register Renaming

- Give processor more registers than specified by the ISA
  - temporarily map ISA registers ("logical" or "architected" registers) to the *physical* registers to avoid overwrites
- Components:
  - mapping mechanism
  - physical registers
    - allocated vs. free registers
    - allocation/deallocation mechanism

Register Renaming

- Example
  - I3 can not exec before I2 because I3 will overwrite R5
  - I5 can not go before I2 because I2, when it goes, will overwrite R2 with a stale value

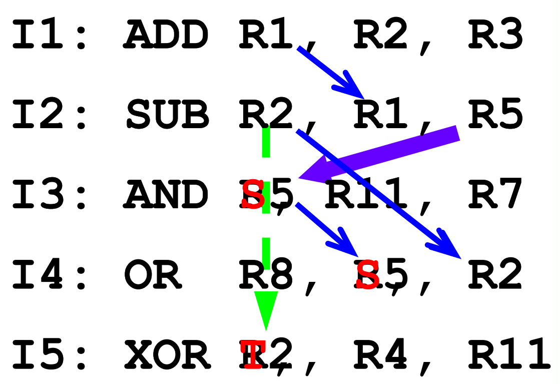
Program code



```
I1: ADD R1, R2, R3
I2: SUB R2, R1, R5
I3: AND R5, R11, R7
I4: OR  R8, R5, R2
I5: XOR R2, R4, R11
```

Register Renaming

- Solution:
  Let’s give I3 temporary name/ location (e.g., S) for the value it produces.
- But I4 uses that value, so we must also change that to S…
- In fact, all uses of R5 from I3 to the next instruction that writes to R5 again must now be changed to S!
- We remove WAW deps in the same way: change R2 in I5 (and subsequent instrs) to T.

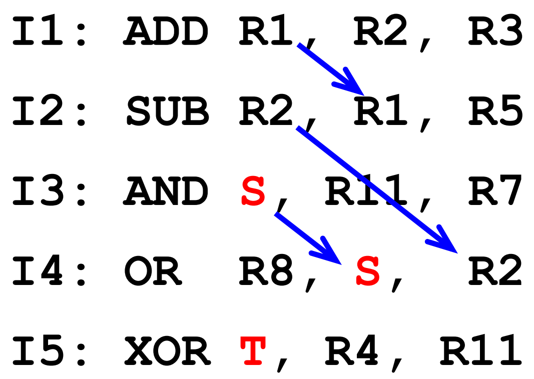


```
I1: ADD R1, R2, R3
I2: SUB R2, R1, R5
I3: AND R5, R11, R7
I4: OR  R8, R5, R2
I5: XOR R2, R4, R11
```

Register Renaming

- Implementation
  - Space for S, T, etc.
  - How do we know when to rename a register?
- Simple Solution
  - Do renaming for every instruction
  - Change the name of a register each time we decode an instruction that will write to it.
  - Remember what name we gave it ☺

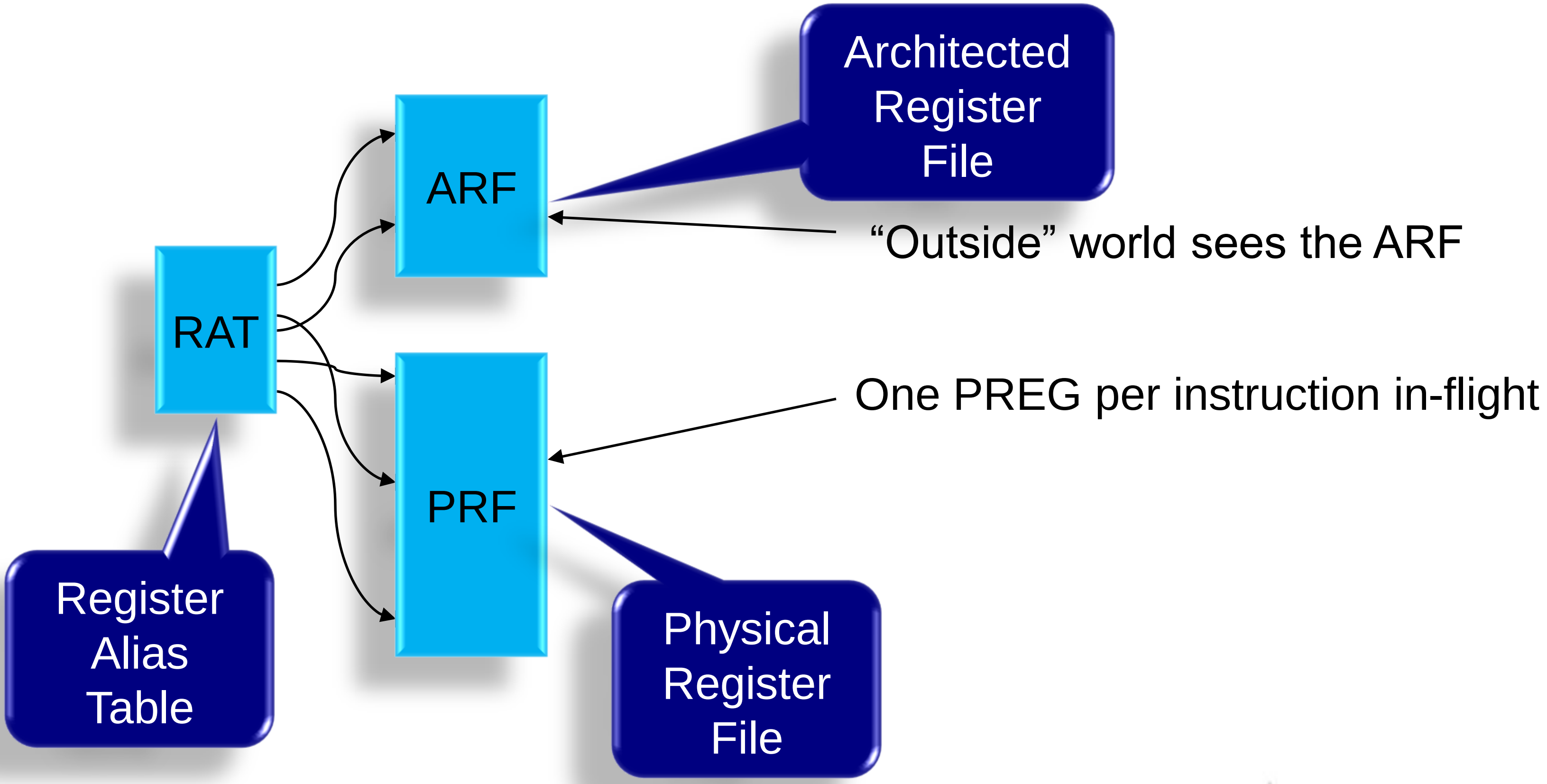
Program code



```
I1: ADD R1, R2, R3
I2: SUB R2, R1, R5
I3: AND S,  R11, R7
I4: OR  R8, S,  R2
I5: XOR T,  R4, R11
```

Register File Organization

- We need some physical structure to store the register values

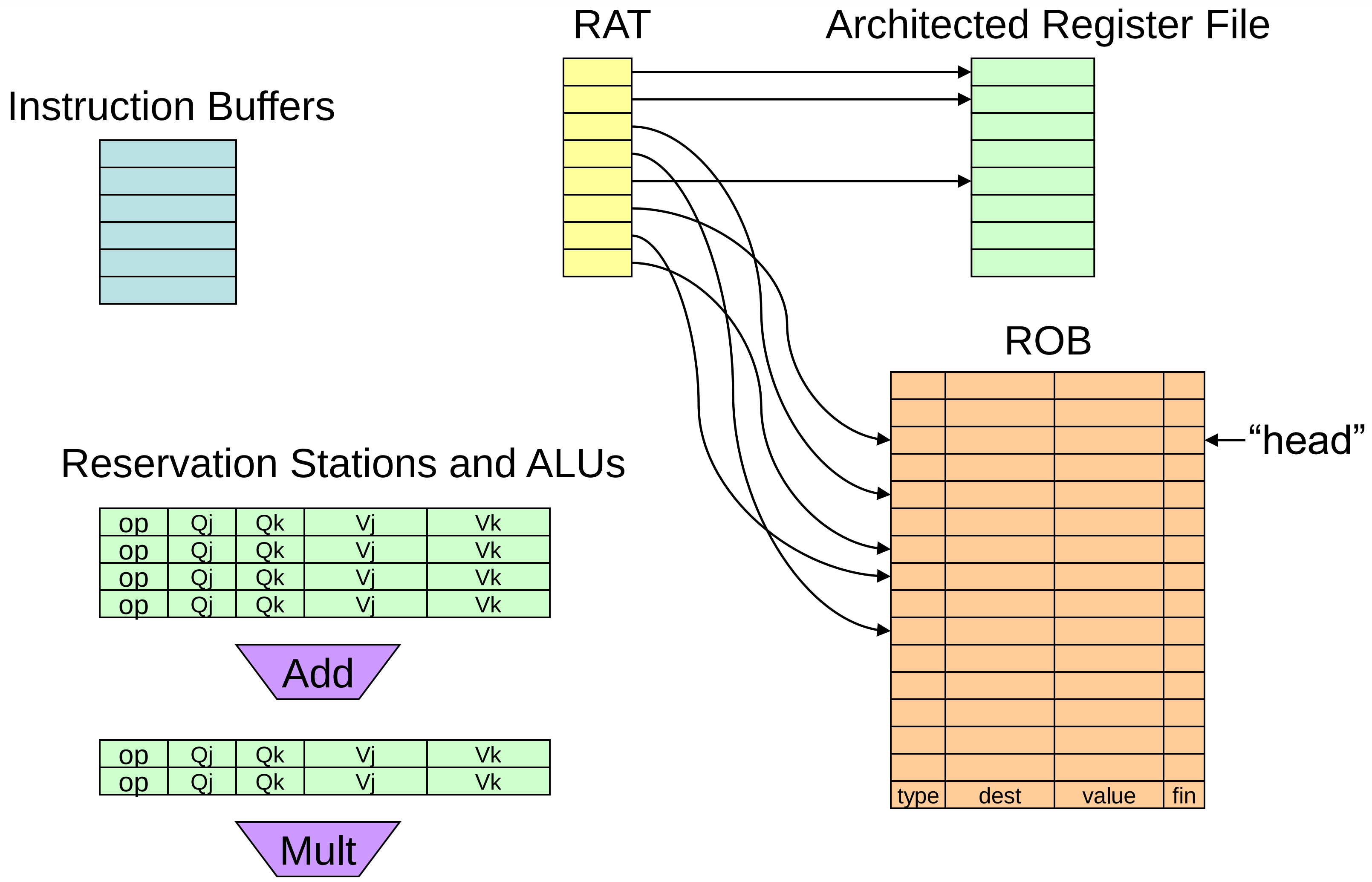

Architected Register File
ARF
"Outside" world sees the ARF
RAT
PRF
One PREG per instruction in-flight
Register Alias Table
Physical Register File

OUT OF ORDER (OOO) EXECUTION

Re-Order Buffer (ROB)

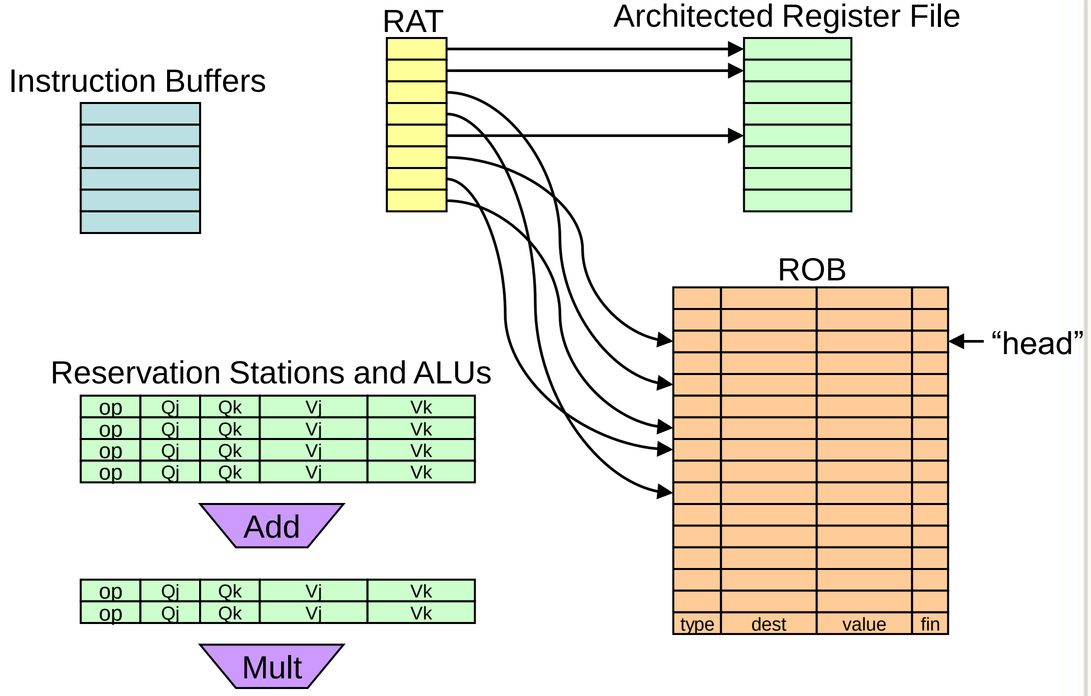
- Separates architected vs. physical registers
- Tracks program order of all in-flight insts
  - Enables in-order completion or “commit”

Hardware Organization



Issue

- Read inst from inst buffer
- Check if resources available:
  - Appropriate RS entry
  - ROB entry
- Read RAT, read (available) sources, update RAT
- Write to RS and ROB



Exec

- Same as before
  - Wait for all operands to arrive
  - Compete to use functional unit
  - Execute!

Write Result

- Broadcast result on CDB
  - (any dependents will grab the value)
- Write result back to your **ROB** entry
  - The ARF holds the “official” register state, which we will only update in program order
  - Mark ready/finished bit in ROB (note that this inst has completed execution)
- Reservation station can be freed.

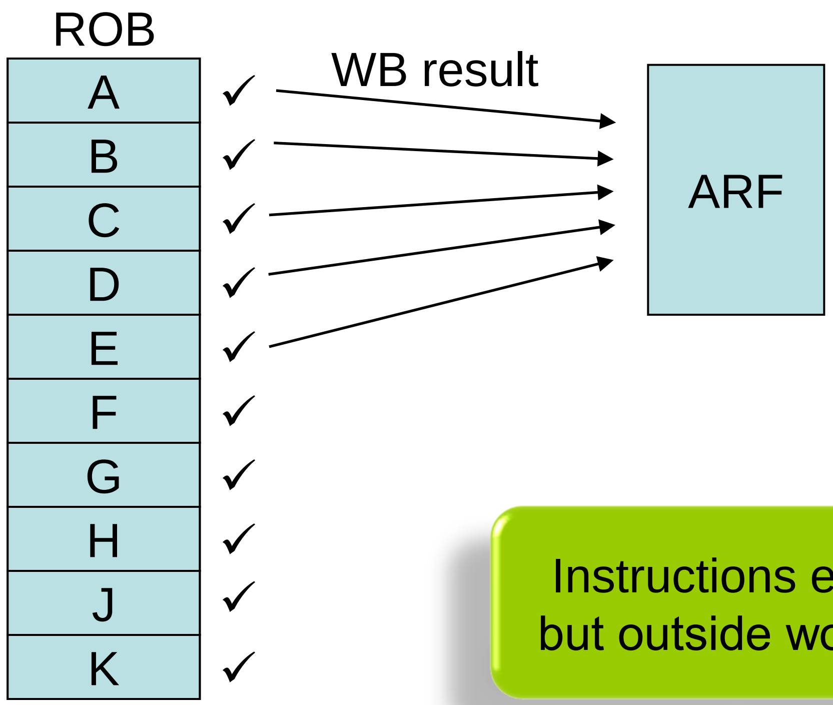
Commit

- When an inst is the oldest in the ROB
  - i.e., ROB-head points to it
- Write result (if ready/finished bit is set)
  - If register producing instruction: write to architected register file
  - If store: write to memory
    - Q: What about load?
- Advance ROB-head to next instruction

- This is what the outside world sees
  - And it’s all in-order

Commit Illustrated

- Make instruction execution "visible" to the outside world
  - "Commit" the changes to the architected state



Outside World "sees":

A executed
B executed
C executed
D executed
E executed

Instructions execute out of program order, but outside world still "believes" it's in-order

SMT

Multithreaded Processors

- Single thread in superscalar execution: dependences cause most of stalls
- Idea: when one thread stalled, other can go
- Different granularities of multithreading
  - Coarse MT: can change thread every few cycles
  - Fine MT: can change thread every cycle
  - Simultaneous Multithreading (SMT)
    - Instrs from different threads even in the same cycle
    - AKA **Hyperthreading**

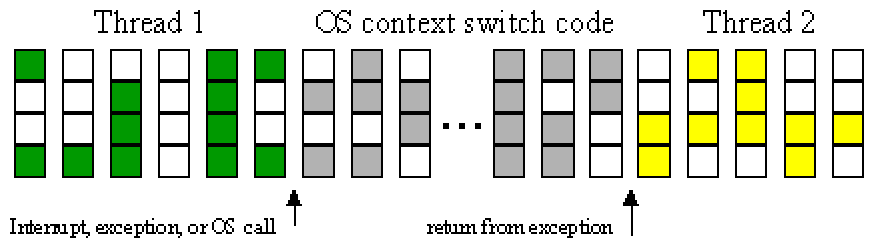
Simultaneous Multi-Threading

- Uni-Processor: 4-6 wide, lucky if you get 1-2 IPC
  - poor utilization
- SMP: 2-4 CPUs, but need independent tasks
  - else poor utilization as well

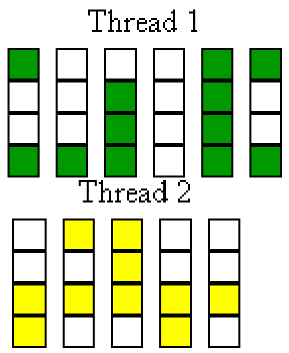
- SMT: Idea is to use a single large uni-processor as a multi-processor

SMT (2)

Regular CPU

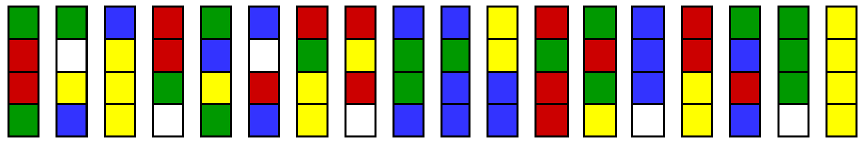


CMP



2x HW Cost

SMT (4 threads)



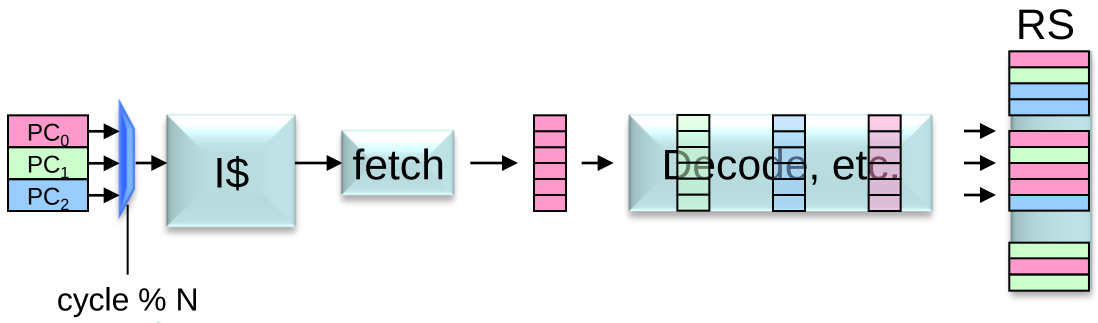
Approx 1x HW Cost

Overview of SMT Hardware Changes

- For an N-way (N threads) SMT, we need:
  - Ability to fetch from N threads
  - N sets of architectural registers (including PCs)
  - N rename tables (RATs)
  - N virtual memory spaces
  - Front-end: branch predictor?: no, RAS? :yes
- But we don’t need to replicate the entire OOO execution engine (schedulers, execution units, bypass networks, ROBs, etc.)

SMT Fetch

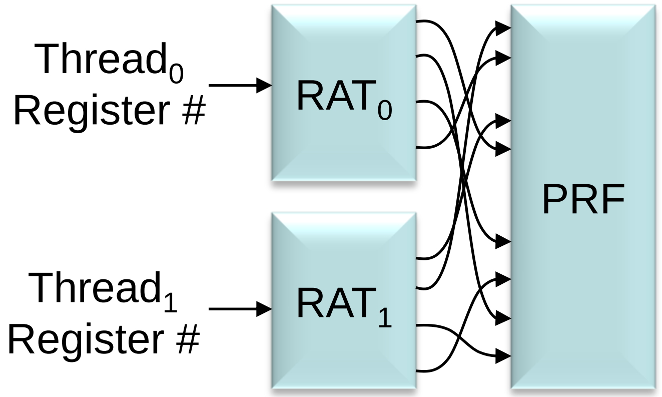
- Multiplex the Fetch Logic



Can do simple round-robin between active threads, or favor some over the others based on how much each is stalling relative to the others

SMT Rename

- Thread #1's R12 != Thread #2's R12
  - separate name spaces
  - need to disambiguate



SMT Issue, Exec, Bypass, …

- No change needed

After Renaming

Thread 0:

Add R1 = R2 + R3
Sub R4 = R1 – R5
Xor R3 = R1 ^ R4
Load R2 = 0[R3]

Thread 1:

Add R1 = R2 + R3
Sub R4 = R1 – R5
Xor R3 = R1 ^ R4
Load R2 = 0[R3]

Thread 0:

Add T12 = T20 + T8
Sub T19 = T12 – T16
Xor T14 = T12 ^ T19
Load T23 = 0[T14]

Thread 1:

Add T17 = T29 + T3
Sub T5 = T17 – T2
Xor T31 = T17 ^ T5
Load T25 = 0[T31]

Shared RS Entries

Sub T5 = T17 – T2
Add T12 = T20 + T8
Load T25 = 0[T31]
Xor T14 = T12 ^ T19
Load T23 = 0[T14]
Sub T19 = T12 – T16
Xor T31 = T17 ^ T5
Add T17 = T29 + T3

SMT Commit

- Register File Management
  - ARF/PRF organization
    - need one ARF per thread

- Need to maintain interrupts, exceptions, faults on a per-thread basis
  - like OOO needs to appear to outside world that it is in-order, SMT needs to appear as if it is actually N CPUs