EECS 252 Graduate Computer Architecture

Lec 11 – Mid Term Review

David Culler
Electrical Engineering and Computer Sciences
University of California, Berkeley

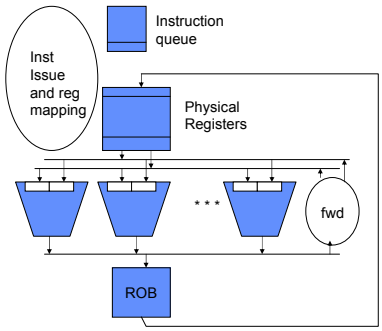
http://www.eecs.berkeley.edu/~culler
http://www-inst.eecs.berkeley.edu/~cs252

Review Exercise

- The 1+X "accumulator" based ISA never seems to go away because of its "minimal" processor state – witness the longevity of the 8051
- You are given the task of designing a "high performance 8051". Having learned about the separation of architected state and microarchitecture, you are ready to attack the problem. A simple analysis suggests that 8051 code has very strong sequential dependences. You will need to use serious instruction lookahead, branch prediction, and register renaming to get at the ILP.
- Assume a MIPS 10K-like data path with multiple function units, lots of physical registers. You need to design the instruction issue and register mapping logic to get ILP out of this beast.
- When is a physical register available for reuse?

Solution Framework

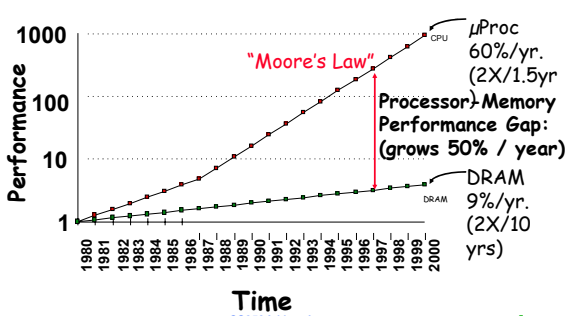
- ISA?
- Typical sequence
- Dependences
- Names?
- Mapping
- Free



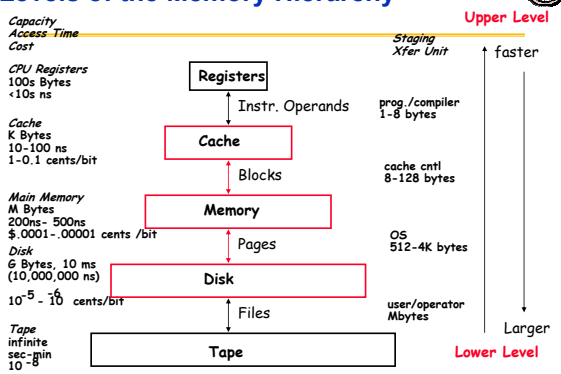
Review of Memory Hierarchy that we skipped

Recap: Who Cares About the Memory Hierarchy?

Processor-DRAM Memory Gap (latency)



Levels of the Memory Hierarchy



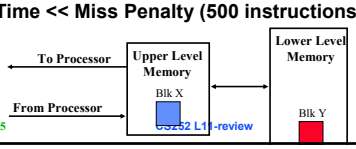
The Principle of Locality

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
- Last 15 years, HW relied on locality for speed

It is a property of programs which is exploited in machine design.

Memory Hierarchy: Terminology

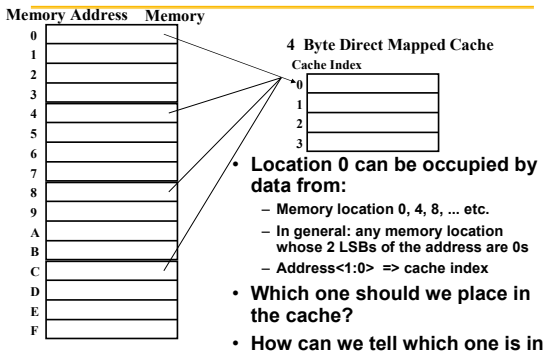
- Hit: data appears in some block in the upper level (example: Block X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieve from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor
- Hit Time << Miss Penalty (500 instructions on 21264!)



Cache Measures

- *Hit rate*: fraction found in that level
  - So high that usually talk about *Miss rate*
  - Miss rate fallacy: as MIPS to CPU performance, miss rate to average memory access time in memory
- Average memory-access time
  = Hit time + Miss rate x Miss penalty
  (ns or clocks)
- *Miss penalty*: time to replace a block from lower level, including time to replace in CPU
  - *access time*: time to lower level
    = f(latency to lower level)
  - *transfer time*: time to transfer block
    =f(BW between upper & lower levels)

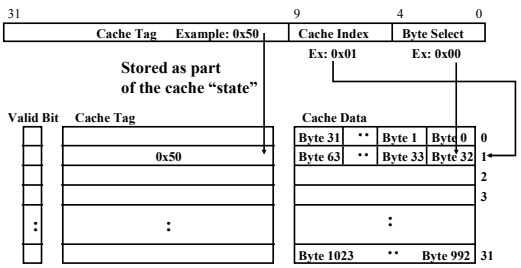
Simplest Cache: Direct Mapped



- Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ... etc.
  - In general: any memory location whose 2 LSBs of the address are 0s
  - Address<1:0> => cache index
- Which one should we place in the cache?
- How can we tell which one is in the cache?

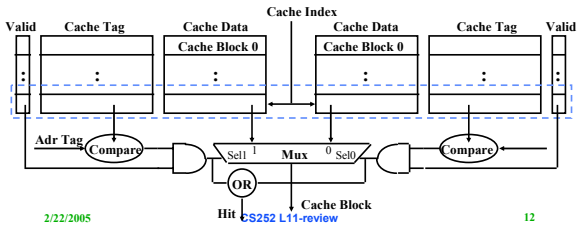
1 KB Direct Mapped Cache, 32B blocks

- For a 2 ** N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2 ** M)



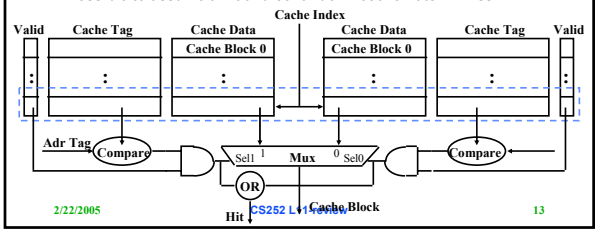
Two-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operates in parallel (N typically 2 to 4)
- Example: Two-way set associative cache
  - Cache Index selects a "set" from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result



Disadvantage of Set Associative Cache

- N-way Set Associative Cache v. Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss
- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.

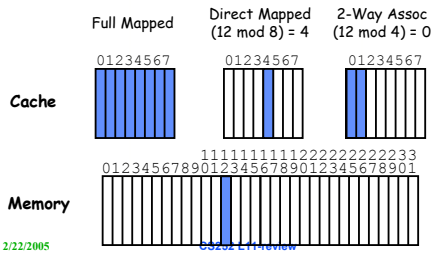


4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level? *(Block placement)*
- Q2: How is a block found if it is in the upper level? *(Block identification)*
- Q3: Which block should be replaced on a miss? *(Block replacement)*
- Q4: What happens on a write? *(Write strategy)*

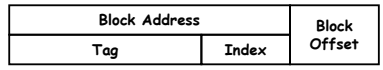
Q1: Where can a block be placed in the upper level?

- Block 12 placed in 8 block cache:
  - Fully associative, direct mapped, 2-way set associative
  - S.A. Mapping = Block Number Modulo Number Sets



Q2: How is a block found if it is in the upper level?

- Tag on each block
  - No need to check index or block offset
- Increasing associativity shrinks index, expands tag



Q3: Which block should be replaced on a miss?

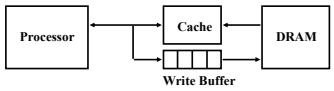
- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

Assoc:	2-way		4-way		8-way	
Size	LRU	Ran	LRU	Ran	LRU	Ran
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Q4: What happens on a write?

- ***Write through***—The information is written to both the block in the cache and to the block in the lower-level memory.
- ***Write back***—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each?
  - WT: read misses cannot result in writes
  - WB: no repeated writes to same location
- WT always combined with write buffers so that don't wait for lower level memory

Write Buffer for Write Through



- A Write Buffer is needed between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
- Memory system designer's nightmare:
  - Store frequency (w.r.t. time) -> 1 / DRAM write cycle
  - Write buffer saturation

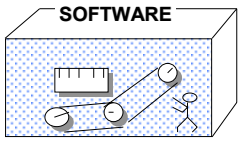
Impact of Memory Hierarchy on Algorithms

- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?
- "The Influence of Caches on the Performance of Sorting" by A. LaMarca and R.E. Ladner. *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, January, 1997, 370-379.
- Quicksort: fastest comparison based sorting algorithm when all keys fit in memory
- Radix sort: also called "linear time" sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys
- For Alphastation 250, 32 byte blocks, direct mapped L2 2MB cache, 8 byte keys, from 4000 to 4000000

Key topics firehose…

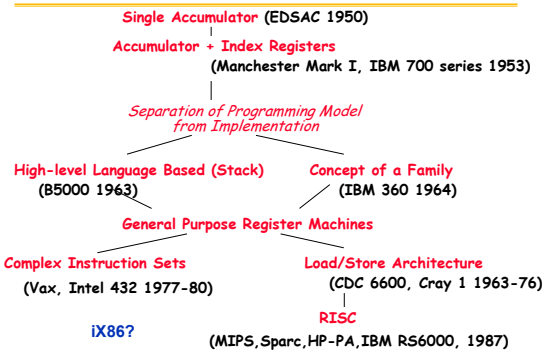
Instruction Set Architecture

... the attributes of a [computing] system as seen by the programmer, *i.e.* the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.
– Amdahl, Blaaw, and Brooks, 1964

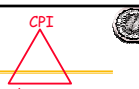


-- Organization of Programmable Storage
-- Data Types & Data Structures: Encodings & Representations
-- Instruction Formats
-- Instruction (or Operation Code) Set
-- Modes of Addressing and Accessing Data Items and Instructions
-- Exceptional Conditions

Evolution of Instruction Sets



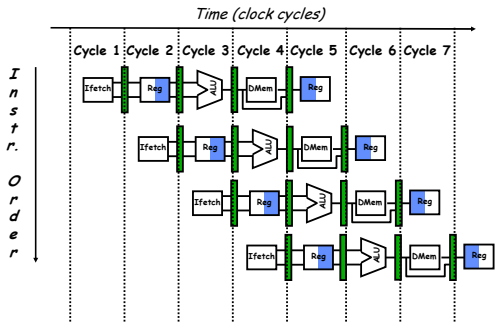
Components of Performance



$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

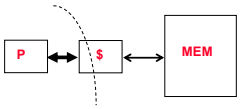
	Inst Count	CPI	Clock Rate
Program	X		
Compiler	X	(X)	
Inst. Set.	X	X	
Organization		X	X
Technology			X

Pipelined Instruction Execution

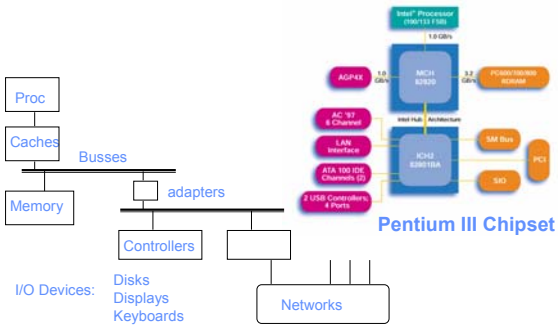


The Principle of Locality

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
- Last 30 years, HW relied on locality for speed



System Organization: It's all about communication



Amdahl's Law

$$ExTime_{new} = ExTime_{old} \times \left[\left(1 - Fraction_{enhanced}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right]$$

$$Speedup_{overall} = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{\left(1 - Fraction_{enhanced}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

Best you could ever hope to do:

$$Speedup_{maximum} = \frac{1}{\left(1 - Fraction_{enhanced}\right)}$$

Cycles Per Instruction (Throughput)

"Average Cycles per Instruction"

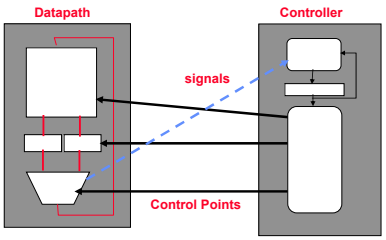
CPI = (CPU Time * Clock Rate) / Instruction Count
= Cycles / Instruction Count

$$CPU\ time = Cycle\ Time \times \sum_{j=1}^{n} CPI_j \times I_j$$

$$CPI = \sum_{j=1}^{n} CPI_j \times F_j \quad \text{where } F_j = \frac{I_j}{\text{Instruction Count}}$$

"Instruction Frequency"

Datapath vs Control



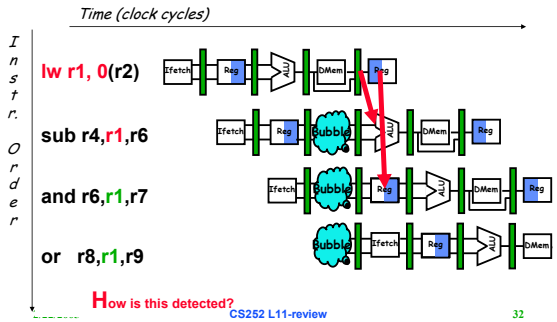
- Datapath: Storage, FU, interconnect sufficient to perform the desired functions
  - Inputs are Control Points
  - Outputs are signals
- Controller: State machine to orchestrate operation on the data path
  - Based on desired function and signals

Pipelining is not quite that easy!

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

Data Hazard Even with Forwarding

Figure 3.13, Page 154



Speed Up Equation for Pipelining

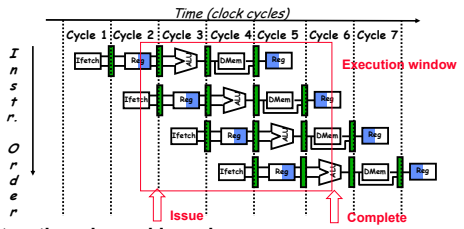
$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

For simple RISC pipeline, CPI = 1:

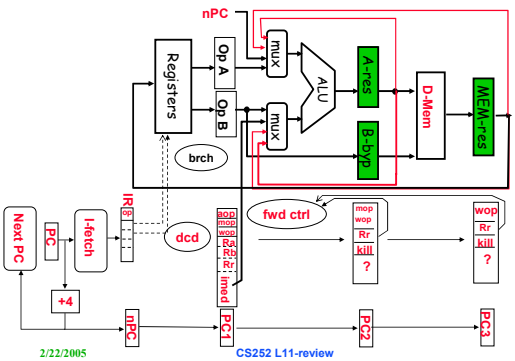
$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

Ordering Properties of basic inst. pipeline



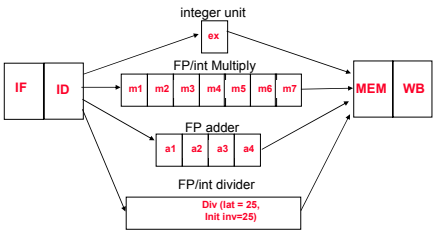
- Instructions issued in order
- Operand fetch is stage 2 => operand fetched in order
- Write back in stage 5 => no WAW, no WAR hazards
- Common pipeline flow => operands complete in order
- Stage changes only at "end of instruction"

Control Pipeline



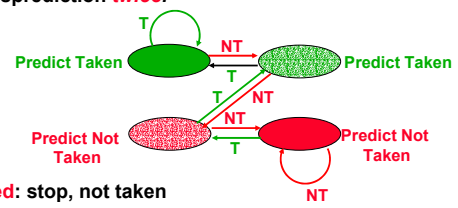
Typical "simple" Pipeline

- Example: MIPS R4000



2-bit Dynamic Branch Prediction
(J. Smith, 1981)

- 2-bit scheme where change prediction only if get misprediction *twice*:



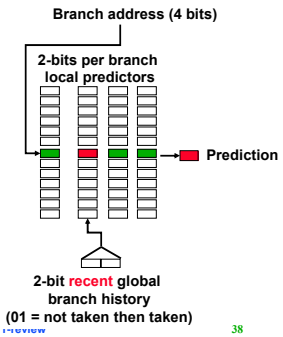
- Red: stop, not taken
- Green: go, taken
- Adds *hysteresis* to decision making process
- Generalize to n-bit saturating counter

Correlating Branches

Idea: taken/not taken of recently executed branches is related to behavior of next branch (as well as the history of that branch behavior)

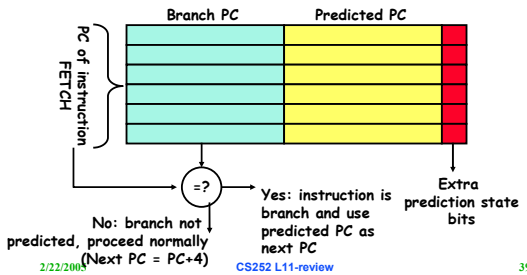
- Then behavior of recent branches selects between, say, 4 predictions of next branch, updating just that prediction

- (2,2) predictor: 2-bit global, 2-bit local



Need Address at Same Time as Prediction

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can't use wrong branch address (Figure 3.19, 3.20)

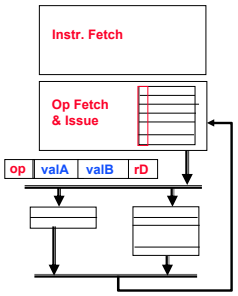


Pipelining with Reg. Reservations

- Assumptions
  1. Multiple pipelined function units of different latency
     - » able to accept operations at issue rate
     - » may be exceptions (e.g., divide)
  2. Issue instructions in order
  3. Operand fetch in order
  4. Completion out of order
     - » short ops may bypass long ones
  5. Some shared resources (e.g., reg write port)
- Implications
  - WAR hazard still resolved by pipeline flow (2 & 3)
  - RAW, WAW, and structural still present
- Design philosophy (ala Cray)
  - Resolve hazards as instruction is issued into pipeline
  - Pipeline is non-blocking

Hazard Resolution

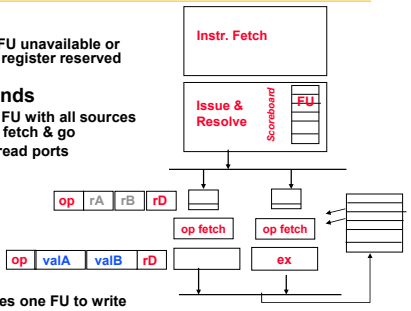
- Structural
  - Op code => resource usage
  - Check resource resv
  - Set on issue
- Data
  - Add reservation bit one each register
  - Check RegRsv for source and destination registers
  - Hold issue till clear
  - Set bit on destination register
  - Clear bit on dest reg. Write
- Questions:
  - Forwarding?



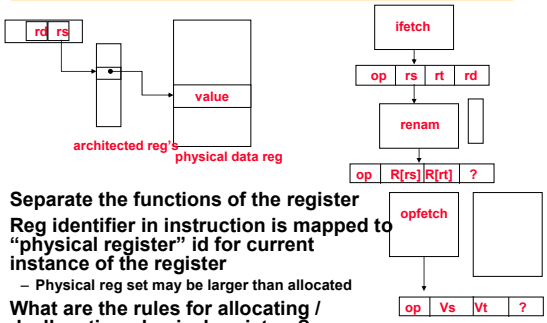
Motorola 88000 "scoreboard" [sic]

Scoreboard Operation

- Issue
  - Hold while FU unavailable or destination register reserved (by FU *f*)
- Read operands
  - SB informs FU with all sources available to fetch & go
  - Limited by read ports
- Write back
  - SB schedules one FU to write
  - Waits no FU waiting to fetch (old version) of reg



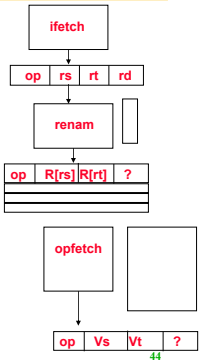
Register Renaming (less Conceptual)



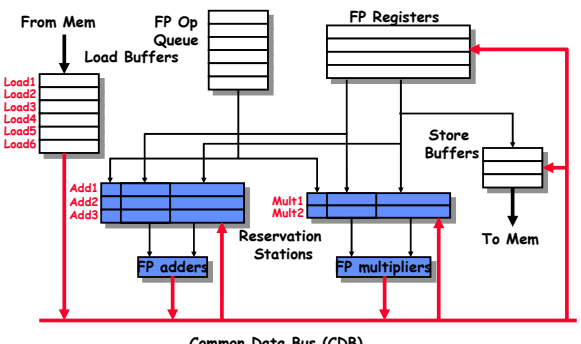
- Separate the functions of the register
- Reg identifier in instruction is mapped to "physical register" id for current instance of the register
  - Physical reg set may be larger than allocated
- What are the rules for allocating / deallocating physical registers?

Reg renaming

- Source Reg s:
  - physical reg P=R[s]
- Destination reg d:
  - Old physical register R[d] "terminates"
  - R[d] :=get_free
- Free physical register when
  - No longer referenced by any architected register (terminated)
  - No incomplete instructions waiting to read it
    - » Easy with in-order
    - » Out of order?



Tomasulo Organization



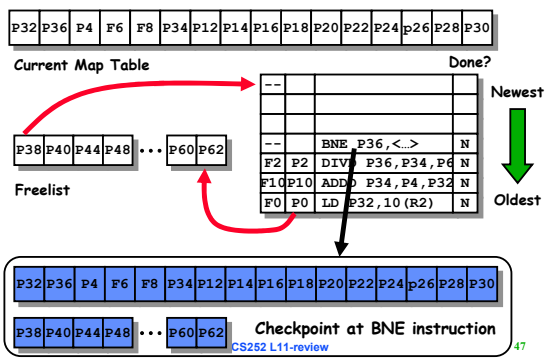
Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue
   If reservation station free (no structural hazard), control issues instr & sends operands (renames registers).
2. Execution—operate on operands (EX)
   When both operands ready then execute; if not ready, watch Common Data Bus for result
3. Write result—finish execution (WB)
   Write on Common Data Bus to all awaiting units; mark reservation station available

- Normal data bus: data + destination ("go to" bus)
- Common data bus: data + source ("come from" bus)
  - 64 bits of data + 4 bits of Functional Unit source address
  - Write if matches expected Functional Unit (produces result)
  - Does the broadcast

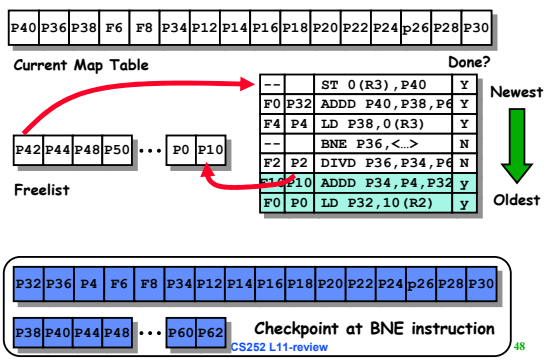
Explicit register renaming:

R10000 Freelist Management

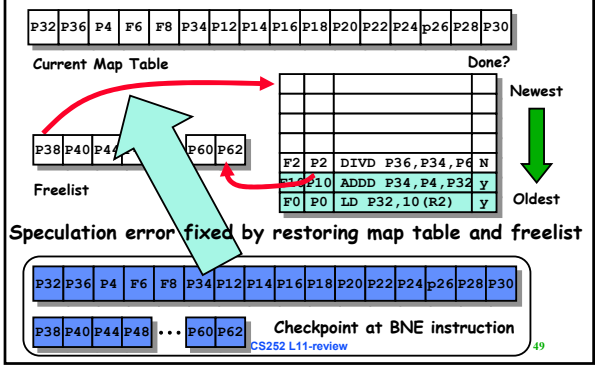


Explicit register renaming:

R10000 Freelist Management



Explicit register renaming:
R10000 Freelist Management



Problems with scalar approach to ILP extraction

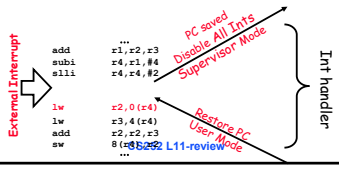
- Limits to conventional exploitation of ILP:
  - *pipelined clock rate*: at some point, each increase in clock rate has corresponding CPI increase (branches, other hazards)
  - *branch prediction*: branches get in the way of wide issue. They are too unpredictable.
  - *instruction fetch and decode*: at some point, its hard to fetch and decode more instructions per clock cycle
  - *register renaming*: Rename logic gets really complicate for many instructions
  - *cache hit rate*: some long-running (scientific) programs have very large data sets accessed with poor locality; others have continuous data streams (multimedia) and hence poor locality

Exception classifications

- *Traps:* relevant to the current process
  - Faults, arithmetic traps, and system "calls"
  - Invoke software on behalf of the currently executing process
- *Interrupts:* caused by asynchronous, outside events
  - I/O devices requiring service (DISK, network)
  - Clock interrupts (real time scheduling)
- *Machine Checks:* caused by serious hardware failure
  - Not always restartable
  - Indicate that bad things have happened.
    - » Non-recoverable ECC error
    - » Machine room fire
    - » Power outage

Precise Interrupts/Exceptions

- An interrupt or exception is considered *precise* if there is a single instruction (or interrupt point) for which:
  - All instructions before that have committed their state
  - No following instructions (including the interrupting instruction) have modified any state.
- This means, that you can restart execution at the interrupt point and "get the right answer"
  - Implicit in our previous example of a device interrupt:
    - » Interrupt point is at first lw instruction



Precise interrupt point may require multiple PCs

```
        addi  r4,r3,#4
        sub   r1,r2,r3
PC:     bne   r1,there
PC+4:   and   r2,r3,r5
        <other insts>
```

⇐ Interrupt point described as <PC,PC+4>

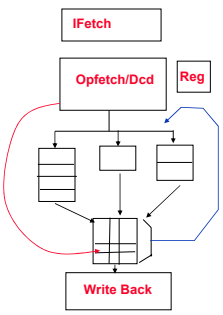
```
        addi  r4,r3,#4
        sub   r1,r2,r3
PC:     bne   r1,there
PC+4:   and   r2,r3,r5
        <other insts>
```

⇐ Interrupt point described as:
<PC+4,there> (branch was taken)
or
<PC+4,PC+8> (branch was not taken)

- On SPARC, interrupt hardware produces "pc" and "npc" (next pc)
- On MIPS, only "pc" – must fix point in software

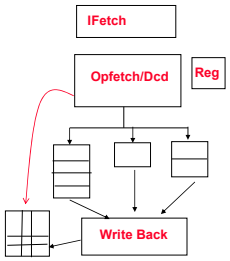
Reorder Buffer + Forwarding + Speculation

- Idea:
  - Issue branch into ROB
  - Mark with prediction
  - Fetch and issue predicted instructions speculatively
  - Branch must resolve before leaving ROB
  - Resolve correct
    - » Commit following instr
  - Resolve incorrect
    - » Mark following instr in ROB as invalid
    - » Let them clear



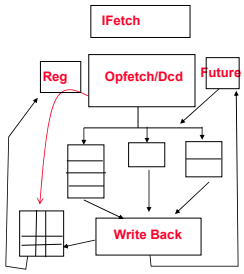
History File

- Maintain issue order, like ROB
- Each entry records dest reg and old value of dest. Register
  - What if old value not available when instruction issues?
- FUs write results into register file
  - Forward into correct entry in history file
- When exception reaches head
  - Restore architected registers from tail to head

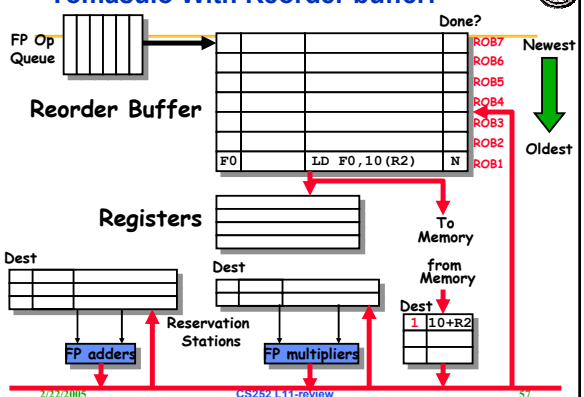


Future file

- Idea
  - Arch registers reflect state at commit point
  - Future register reflect whatever instructions have completed
  - On WB update future
  - On commit update arch
  - On exception
    - » Discard future
    - » Replace with arch
      - Dest w/I ROB

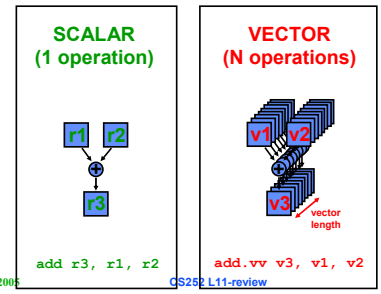


Tomasulo With Reorder buffer:



Alternative Model:Vector Processing

- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"



What needs to be specified in a Vector Instruction Set Architecture?

- ISA in general
  - Operations, Data types, Format, Accessible Storage, Addressing Modes, Exceptional Conditions
- Vectors
  - Operations
  - Data types (Float, int, V op V, S op V)
  - Format
  - Source and Destination Operands
    - » Memory?, register?
  - Length
  - Successor (consecutive, stride, indexed, gather/scatter, …)
  - Conditional operations
  - Exceptions

"DLXV" Vector Instructions

Instr.	Operands	Operation	Comment
• ADDV	V1,V2,V3	V1=V2+V3	vector + vector
• ADDSV	V1,F0,V2	V1=F0+V2	scalar + vector
• MULTV	V1,V2,V3	V1=V2xV3	vector x vector
• MULSV	V1,F0,V2	V1=F0xV2	scalar x vector
• LV	V1,R1	V1=M[R1..R1+63]	load, stride=1
• LVWS	V1,R1,R2	V1=M[R1..R1+63*R2]	load, stride=R2
• LVI	V1,R1,V2	V1=M[R1+V2i,i=0..63]	indir.("gather")
• CeqV	VM,V1,V2	VMASKi = (V1i=V2i)?	comp. setmask
• MOV	VLR,R1	Vec. Len. Reg. = R1	set vector length
• MOV	VM,R1	Vec. Mask = R1	set vector mask

Vector Execution Time

- Time = f(vector length, data dependicies, struct. hazards)
- *Initiation rate*: rate that FU consumes vector elements (= number of lanes; usually 1 or 2 on Cray T-90)
- *Convoy*: set of vector instructions that can begin execution in same clock (no struct. or data hazards)
- *Chime*: approx. time for a vector operation
- *m convoys take m chimes*; if each vector length is n, then they take approx. *m* x *n* clock cycles (ignores overhead; good approximation for long vectors)

```
1: LV    V1,Rx    ;load vector X
2: MULV  V2,F0,V1 ;vector-scalar mult.
   LV    V3,Ry    ;load vector Y
3: ADDV  V4,V2,V3 ;add
4: SV    Ry,V4    ;store the result
```

4 convoys, 1 lane, VL=64
=> 4 x 64 = 256 clocks
(or 4 clocks per result)

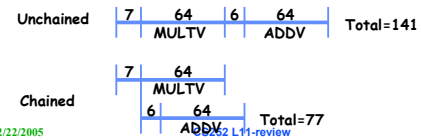
Strip Mining

- Suppose Vector Length > Max. Vector Length (MVL)?
- *Strip mining*: generation of code such that each vector operation is done for a size S to the MVL
- 1st loop do short piece (n mod MVL), rest VL = MVL

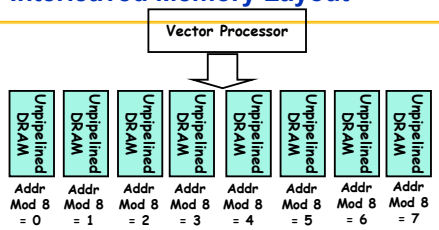
```
low = 1
VL = (n mod MVL)  /*find the odd size piece*/
do 1 j = 0,(n / MVL)  /*outer loop*/
    do 10 i = low,low+VL-1  /*runs for length VL*/
            Y(i) = a*X(i) + Y(i)  /*main operation*/
10  continue
    low = low+VL  /*start of next vector*/
    VL = MVL  /*reset the length to max*/
1   continue
```


Vector Opt #1: Chaining

- Suppose:
  MULV V1,V2,V3
  ADDV V4,V1,V5 ; separate convoy?
- *chaining*: vector register (V1) is not as a single entity but as a group of individual registers, then pipeline forwarding can work on individual elements of a vector
- *Flexible chaining*: allow vector to chain to any other active vector operation => more read/write ports
- As long as enough HW, increases convoy size



Interleaved Memory Layout



- Great for unit stride:
  - Contiguous elements in different DRAMs
  - Startup time for vector operation is latency of single read
- What about non-unit stride?
  - Above good for strides that are relatively prime to 8
  - Bad for: 2, 4
  - Better: prime number of banks...!