

# Caches

Samira Khan

March 21, 2017

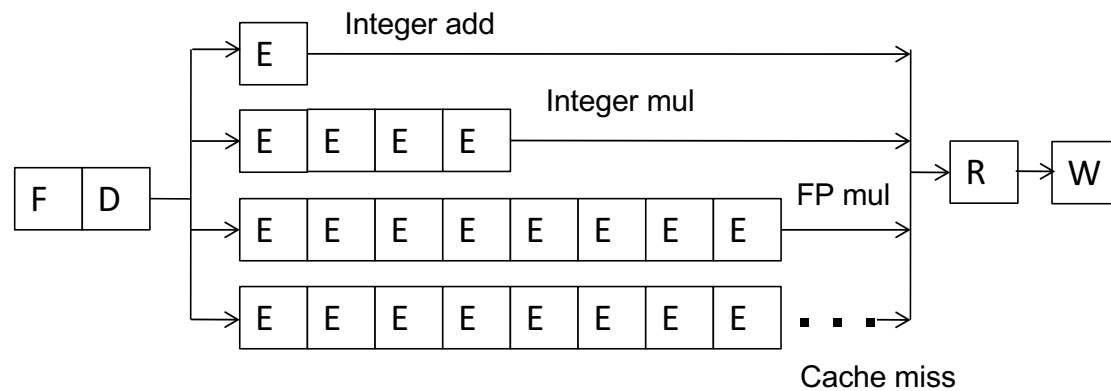
# Agenda

- Logistics
- Review from last lecture
  - Out-of-order execution
- Data flow model
- Superscalar processor
- Caches

# Final Exam

- Combined final exam **7-10PM on Tuesday, 9 May 2017**
- Any conflict?
  - Please fill out the form
  - <https://goo.gl/forms/TVOlvx76N4RiEtC2>
  - Also linked from the schedule page

# AN IN-ORDER PIPELINE



- Problem: A true data dependency stalls dispatch of younger instructions into functional (execution) units
- Dispatch: Act of sending an instruction to a functional unit

# CAN WE DO BETTER?

- What do the following two pieces of code have in common (with respect to execution in the previous design)?

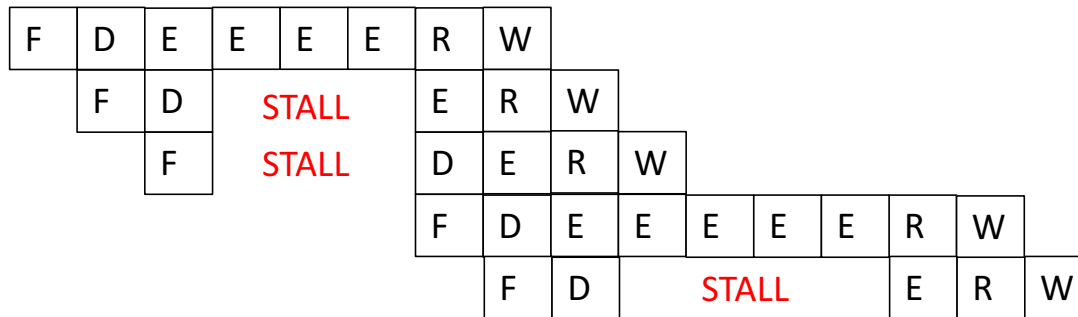
```
IMUL R3 ← R1, R2
ADD  R3 ← R3, R1
ADD  R1 ← R6, R7
IMUL R5 ← R6, R8
ADD  R7 ← R9, R9
```

```
LD   R3 ← R1 (0)
ADD  R3 ← R3, R1
ADD  R1 ← R6, R7
IMUL R5 ← R6, R8
ADD  R7 ← R9, R9
```

- **Answer: First ADD stalls the whole pipeline!**
  - ADD cannot dispatch because its source registers unavailable
  - Later **independent** instructions cannot get executed
- How are the above code portions different?
  - **Answer: Load latency is variable (unknown until runtime)**
  - What does this affect? Think compiler vs. microarchitecture

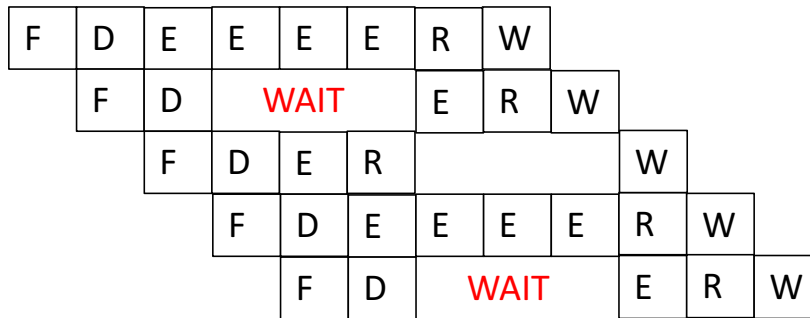
# IN-ORDER VS. OUT-OF-ORDER DISPATCH

- In order dispatch + precise exceptions:



IMUL R3 ← R1, R2  
 ADD R3 ← R3, R1  
 ADD R1 ← R6, R7  
 IMUL R5 ← R6, R8  
 ADD R7 ← R3, R5

- Out-of-order dispatch + precise exceptions:



- 16 vs. 12 cycles

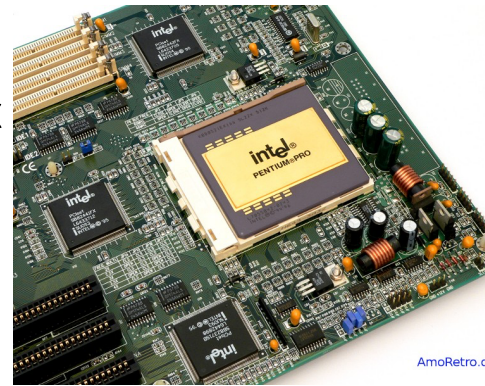
# TOMASULO'S ALGORITHM

- OoO with register renaming invented by Robert Tomasulo
  - Used in IBM 360/91 Floating Point Units
  - Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arithmetic Units,”
  - IBM Journal of R&D, Jan. 1967
- What is the major difference today?
  - **Precise exceptions**: IBM 360/91 did NOT have this
  - Patt, Hwu, Shebanow, “HPS, a new microarchitecture: rationale and introduction,” MICRO 1985.
  - Patt et al., “Critical issues regarding HPS, a high performance microarchitecture,” MICRO 1985.

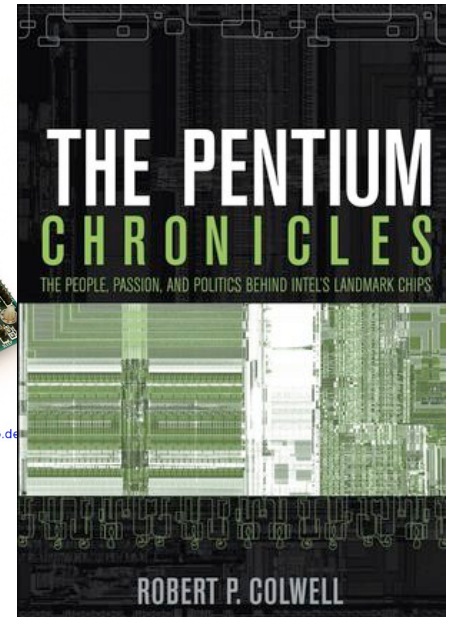


# Out-of-Order Execution \w Precise Exception

- Variants are used in most high-performance processors
  - Initially in Intel Pentium Pro, AMD K5
  - Alpha 21264, MIPS R10000, IBM POWER5, IBM z196, Oracle UltraSPARC T4, ARM Cortex A15



- The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips by [Robert P. Colwell](#)





# Agenda

- Logistics
- Review from last lecture
  - Out-of-order execution
- **Data flow model**
- Superscalar processor
- Caches

# The Von Neumann Model/Architecture

- Also called *stored program computer* (instructions in memory). Two key properties:
- Stored program
  - Instructions stored in a linear memory array
  - **Memory is unified** between instructions and data
    - **The interpretation of a stored value depends on the control signals**  
**When is a value interpreted as an instruction?**
- Sequential instruction processing
  - One instruction processed (fetched, executed, and completed) at a time
  - **Program counter (instruction pointer)** identifies the current instr.
  - **Program counter is advanced sequentially** except for control transfer instructions

# The Dataflow Model (of a Computer)

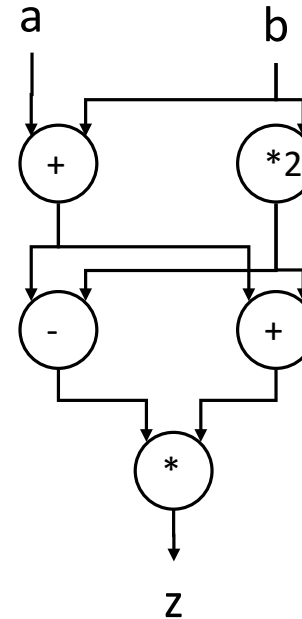
- Von Neumann model: An instruction is fetched and executed in **control flow order**
  - As specified by the **instruction pointer**
  - Sequential unless explicit control flow instruction
- Dataflow model: An instruction is fetched and executed in **data flow order**
  - i.e., when its operands are ready
  - i.e., there is **no instruction pointer**
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies “who” should receive the result
    - An instruction can “fire” whenever all operands are received
  - Potentially many instructions can execute at the same time
    - Inherently more parallel

# Von Neumann vs Dataflow

- Consider a Von Neumann program
  - What is the significance of the program order?
  - What is the significance of the storage locations?

```
v <= a + b;  
w <= b * 2;  
x <= v - w  
y <= v + w  
z <= x * y
```

Sequential

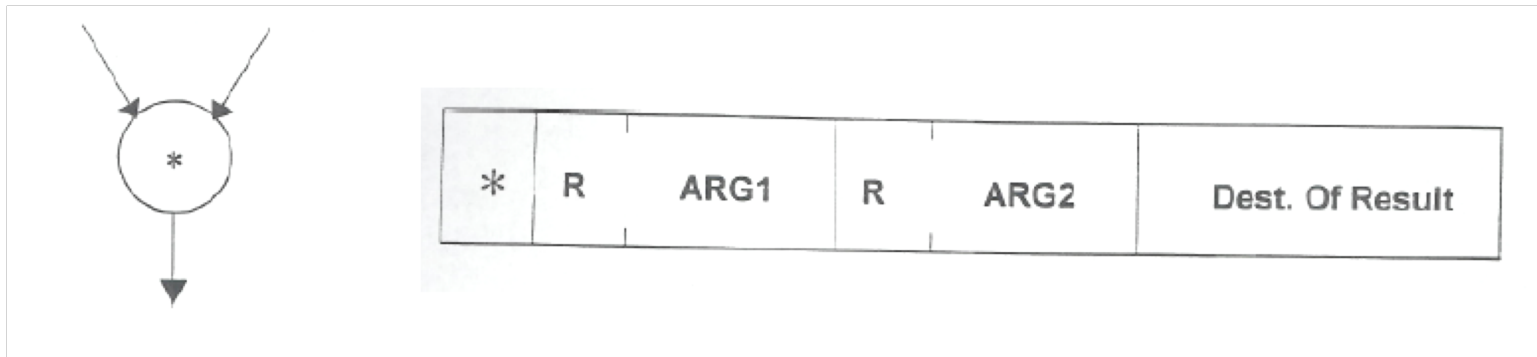


Dataflow

■ Which model is more natural to you as a programmer?

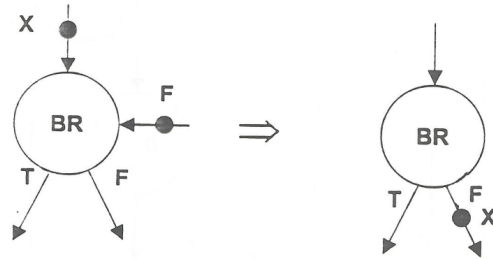
# More on Data Flow

- In a data flow machine, a program consists of data flow nodes
  - A data flow node fires (fetched and executed) when all its inputs are ready
    - i.e. when all inputs have tokens
- Data flow node and its ISA representation

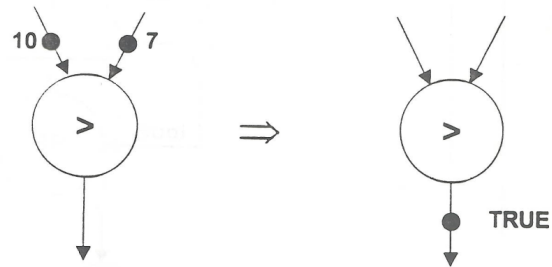


# Data Flow Nodes

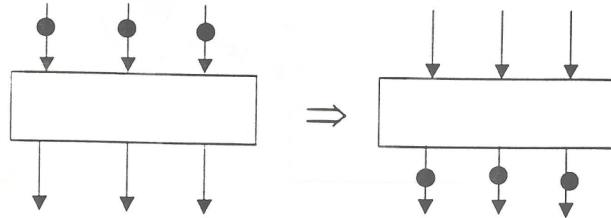
**\*Conditional**



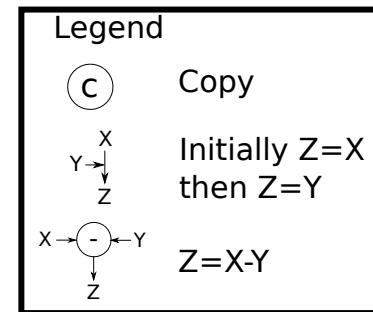
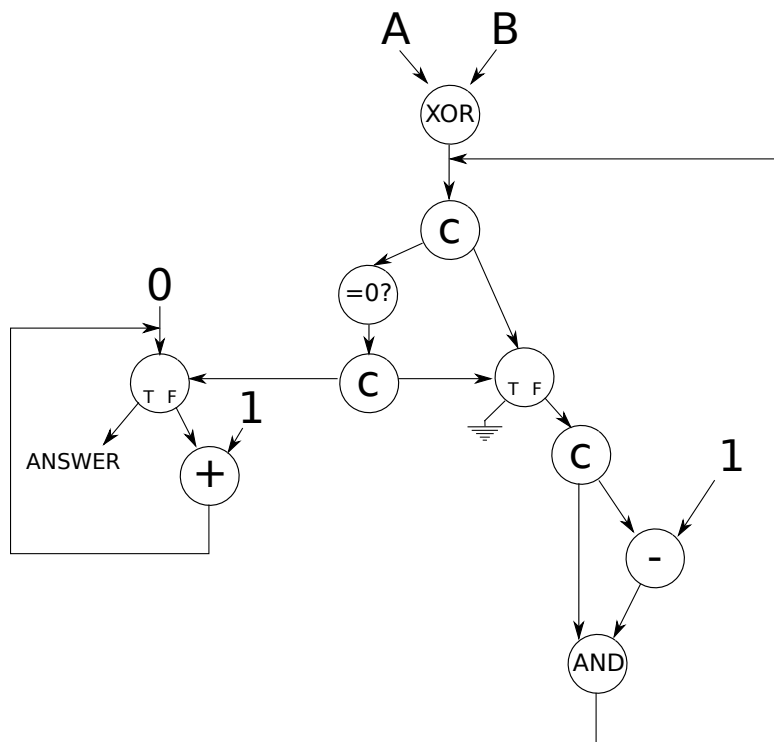
**\*Relational**



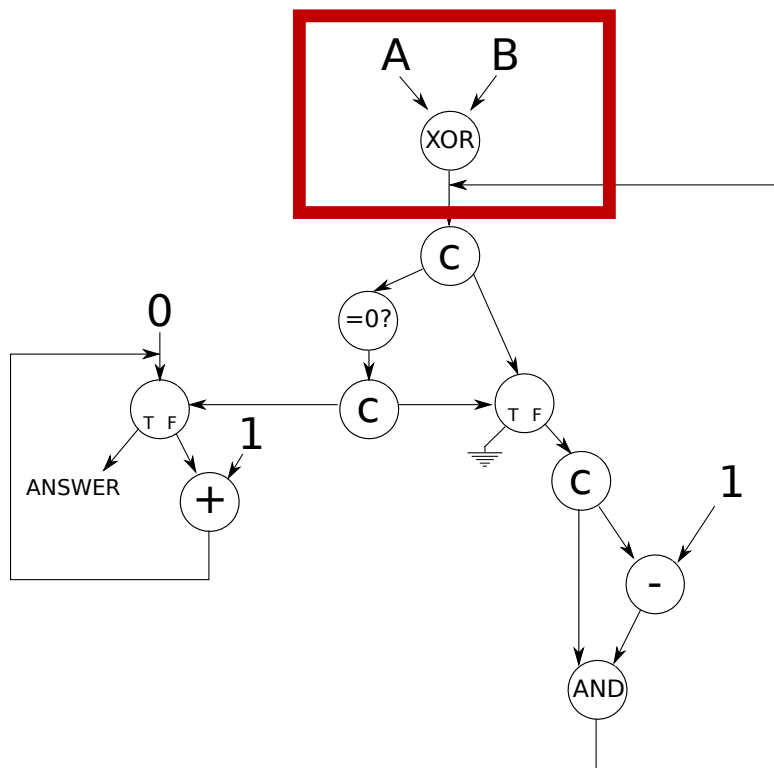
**\*Barrier Synchron**



# An Example



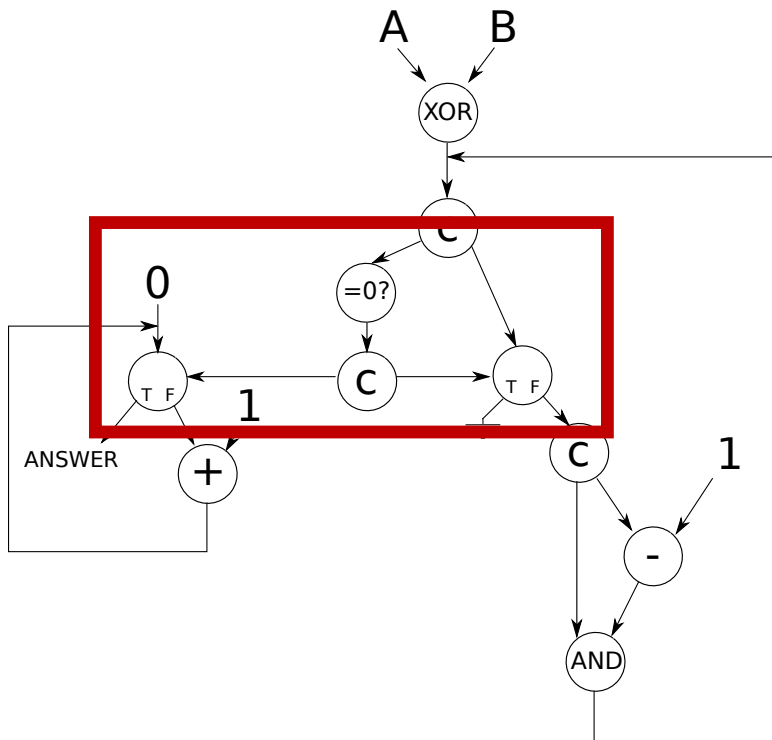
What does this model perform?



val = a ^ b



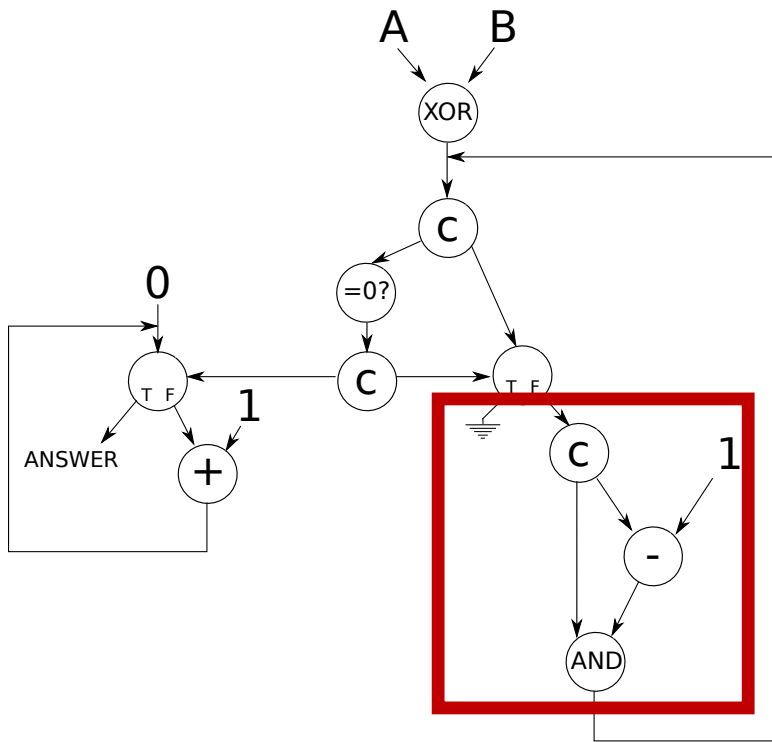
What does this model perform?



val = a ^ b

val != 0

# What does this model perform?

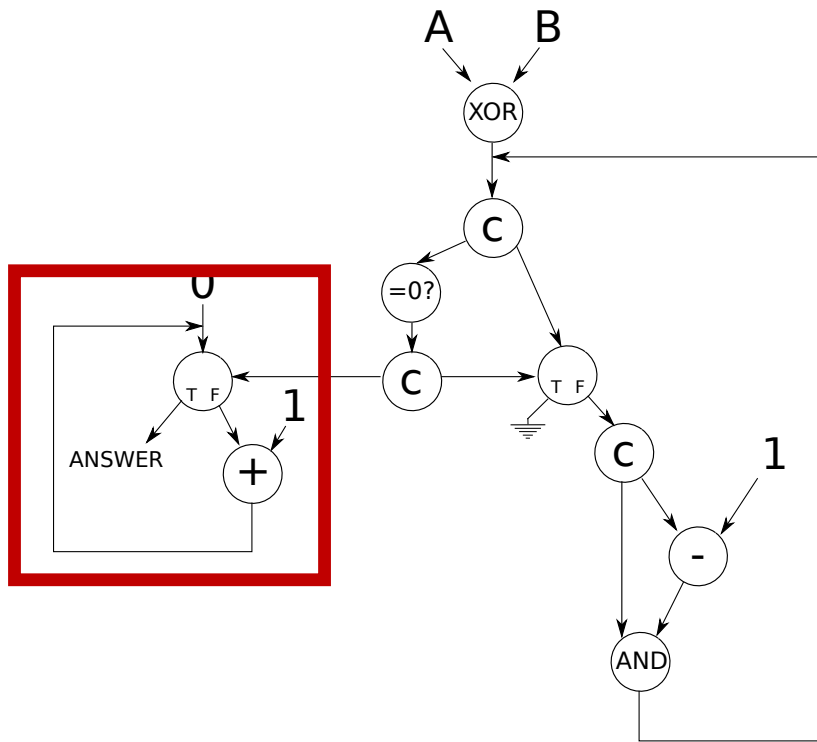


`val = a ^ b`

`val != 0`

`val &= val - 1;`

# What does this model perform?



`val = a ^ b`

`val != 0`

`val &= val - 1;`

`dist = 0`  
`dist++;`

# Hamming Distance

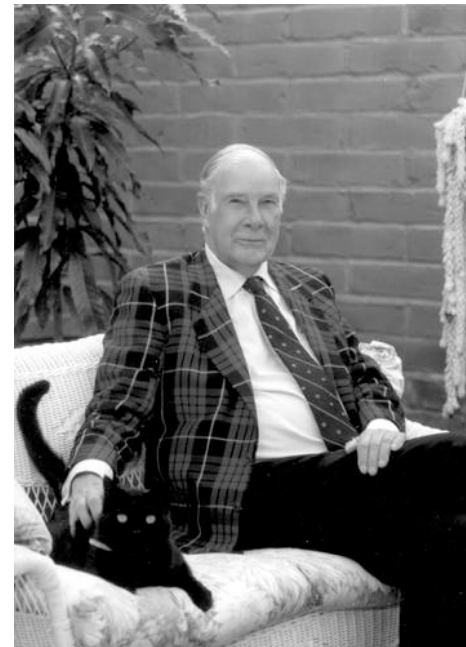
```
int hamming_distance(unsigned a, unsigned b) {  
    int dist = 0;  
    unsigned val = a ^ b; // Count the number of bits set  
    while (val != 0) { // A bit is set, so increment the count and clear the bit  
        dist++; val &= val - 1;  
    }  
    // Return the number of differing bits  
    return dist;  
}
```

# Hamming Distance

- Number of positions at which the corresponding symbols are different.
- The Hamming distance between:
  - "karolin" and "kathrin" is 3
  - **1011101** and **1001001** is 2
  - **2173896** and **2233796** is 3

# RICHARD HAMMING

- Best known for **Hamming Code**
- Won **Turing Award** in 1968
- Was part of the **Manhattan Project**
- Worked in **Bell Labs** for 30 years
- **You and Your Research** is mainly his advice to other researchers
  - Had given the talk many times during his life time
  - <http://www.cs.virginia.edu/~robins/YouAndYourResearch.html>



# Data Flow Advantages/Disadvantages

- Advantages
  - Very good at exploiting **irregular parallelism**
  - Only real dependencies constrain processing
- Disadvantages
  - Debugging difficult (no precise state)
    - Interrupt/exception handling is difficult (what is precise state semantics?)
  - Too much parallelism? (Parallelism control needed)
  - High bookkeeping overhead (tag matching, data storage)
  - Memory locality is not exploited

# OOO EXECUTION: RESTRICTED DATAFLOW

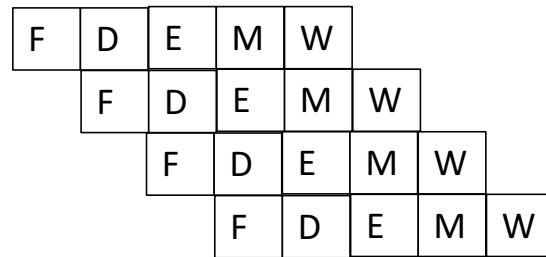
- An out-of-order engine dynamically builds the dataflow graph of a piece of the program
  - which piece?
- The dataflow graph is limited to the instruction window
  - Instruction window: all decoded but not yet retired instructions
- Can we do it for the whole program?
- Why would we like to?
- In other words, how can we have a large instruction window?



# Agenda

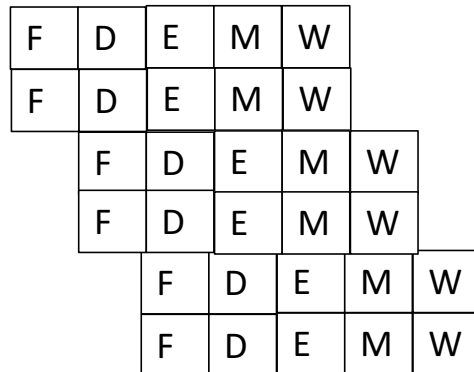
- Logistics
- Review from last lecture
  - Out-of-order execution
- Data flow model
- Superscalar processor
- Caches

# Superscalar Processor



Each instruction still takes 5 cycles, but instructions now complete every cycle:

CPI  $\rightarrow$  1



Each instruction still takes 5 cycles, but instructions now complete every cycle:

CPI  $\rightarrow$  0.5

# Superscalar Processor

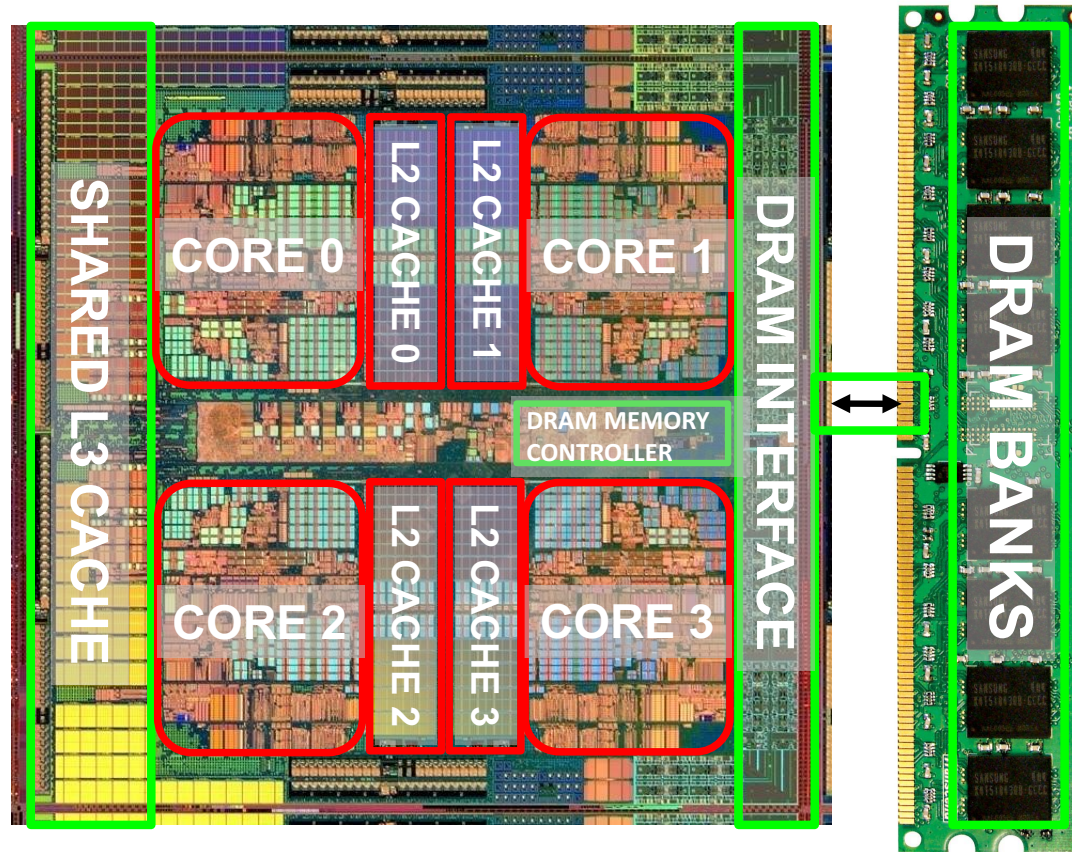
- Ideally: in an n-issue superscalar, n instructions are fetched, decoded, executed, and committed per cycle
- In practice:
  - Data, control, and structural hazards spoil issue flow
  - Multi-cycle instructions spoil commit flow
- Buffers at issue (issue queue) and commit (reorder buffer)
- Decouple these stages from the rest of the pipeline and regularize somewhat breaks in the flow

# Problems?

- Fetch
  - May be located at different cachelines
    - More than one cache lookup is required in the same cycle
  - What if there are branches?
    - Branch prediction is required within the instruction fetch stage
- Decode/Execute
  - Replicate (ok)
- Issue
  - Number of dependence tests increases quadratically (bad)
- Register read/write
  - Number of register ports increases linearly (bad)
- Bypass/forwarding
  - Increases quadratically (bad)

# The Memory Hierarchy

# Memory in a Modern System



# Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

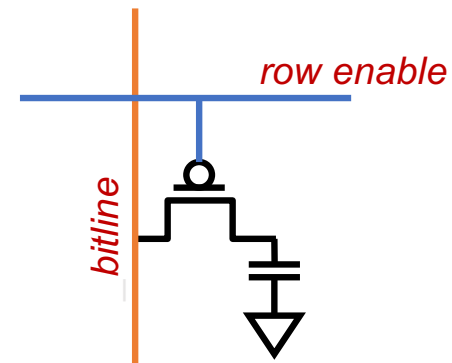
# The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: SRAM vs. DRAM vs. Disk vs. Tape
- Higher bandwidth is more expensive
  - Need more banks, more ports, higher frequency, or faster technology



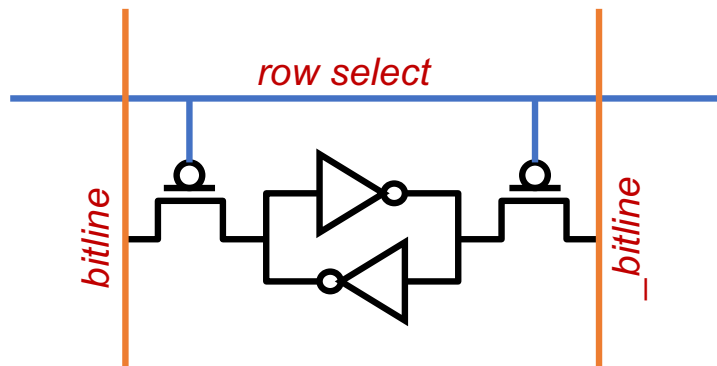
# Memory Technology: DRAM

- Dynamic random access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor
- Capacitor leaks through the RC path
  - DRAM cell loses charge over time
  - DRAM cell needs to be refreshed



# Memory Technology: SRAM

- Static random access memory
- Two cross coupled inverters store a single bit
  - Feedback path enables the stored value to be stable in the “cell”
  - 4 transistors for storage
  - 2 transistors for access



# DRAM vs. SRAM

- DRAM
  - Slower access (capacitor)
  - Higher density (1T 1C cell)
  - Lower cost
  - Requires refresh (power, performance, circuitry)
  - Manufacturing requires putting capacitor and logic together
- SRAM
  - Faster access (no capacitor)
  - Lower density (6T cell)
  - Higher cost
  - No need for refresh
  - Manufacturing compatible with logic process (no capacitor)

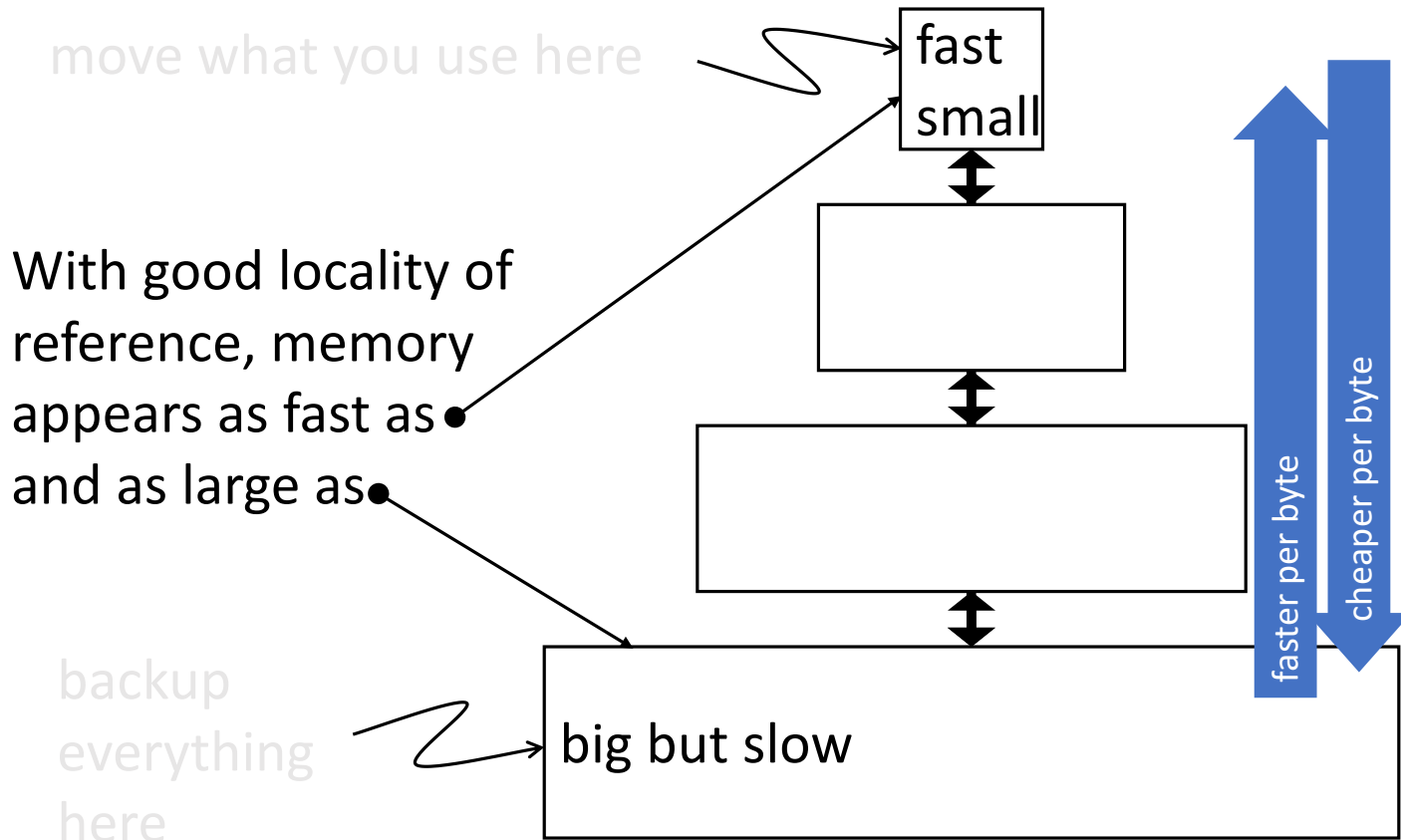
# The Problem

- **Bigger is slower**
  - SRAM, 512 Bytes, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec
- **Faster is more expensive (dollars and chip area)**
  - SRAM, < 10\$ per Megabyte
  - DRAM, < 1\$ per Megabyte
  - Hard Disk < 1\$ per Gigabyte
  - These sample values scale with time
- **Other technologies have their place as well**
  - Flash memory, PC-RAM, MRAM, RRAM (not mature yet)

# Why Memory Hierarchy?

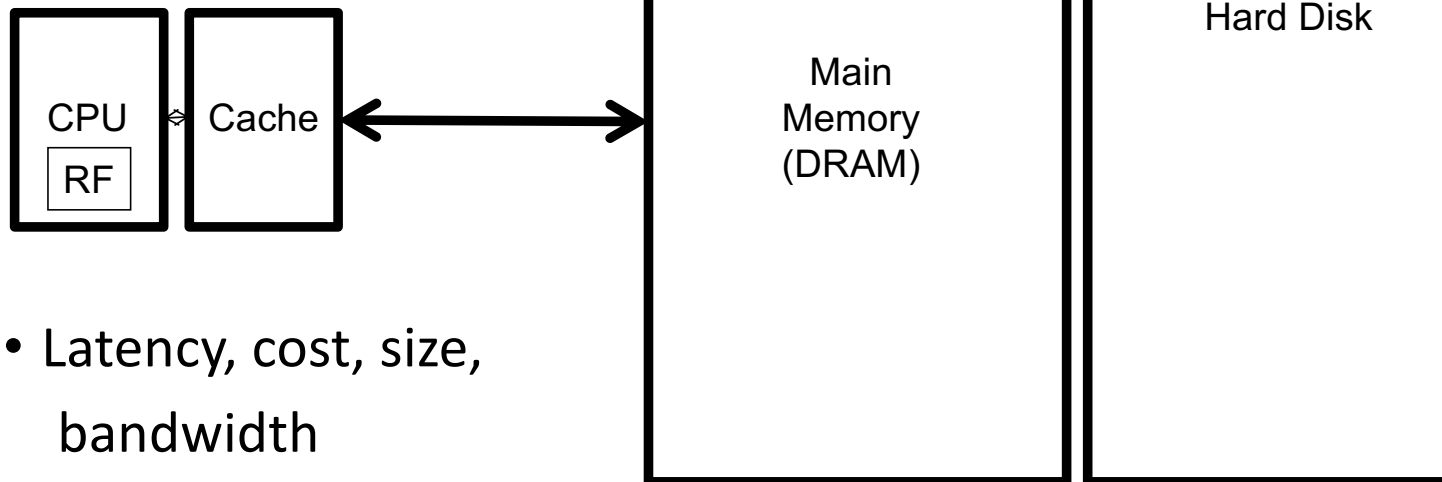
- We want both fast and large
- But we cannot achieve both with a single level of memory
- Idea: **Have multiple levels of storage** (progressively bigger and slower as the levels are farther from the processor) and **ensure most of the data the processor needs is kept in the fast(er) level(s)**

# The Memory Hierarchy



# Memory Hierarchy

- Fundamental tradeoff
  - Fast memory: small
  - Large memory: slow
- Idea: **Memory hierarchy**



- Latency, cost, size, bandwidth

# Locality

- One's recent past is a very good predictor of his/her near future.
- **Temporal Locality**: If you just did something, it is very likely that you will do the same thing again soon
  - since you are here today, there is a good chance you will be here again and again regularly
- **Spatial Locality**: If you did something, it is very likely you will do something similar/related (in space)
  - every time I find you in this room, you are probably sitting close to the same people



# Memory Locality

- A “typical” program has a lot of locality in memory references
  - typical programs are composed of “loops”
- **Temporal**: A program tends to reference the same memory location many times and all within a small window of time
- **Spatial**: A program tends to reference a cluster of memory locations at a time
  - most notable examples:
    - instruction memory references
    - array/data structure references

# Caching Basics: Exploit Temporal Locality

- Idea: Store recently accessed data in automatically managed fast memory (called cache)
- Anticipation: the data will be accessed again soon
- Temporal locality principle
  - Recently accessed data will be again accessed in the near future
  - This is what Maurice Wilkes had in mind:
    - Wilkes, “Slave Memories and Dynamic Storage Allocation,” IEEE Trans. On Electronic Computers, 1965.
    - “The use is discussed of a fast core memory of, say 32000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.”

# Caching Basics: Exploit Spatial Locality

- Idea: **Store addresses adjacent to the recently accessed one in automatically managed fast memory**
  - Logically divide memory into equal size blocks
  - Fetch to cache the accessed block in its entirety
- Anticipation: **nearby data will be accessed soon**
  
- **Spatial locality** principle
  - **Nearby data in memory will be accessed in the near future**
    - E.g., sequential instruction access, array traversal
  - This is what IBM 360/85 implemented
    - 16 Kbyte cache with 64 byte blocks
    - Liptay, “**Structural aspects of the System/360 Model 85 II: the cache,**” IBM Systems Journal, 1968.

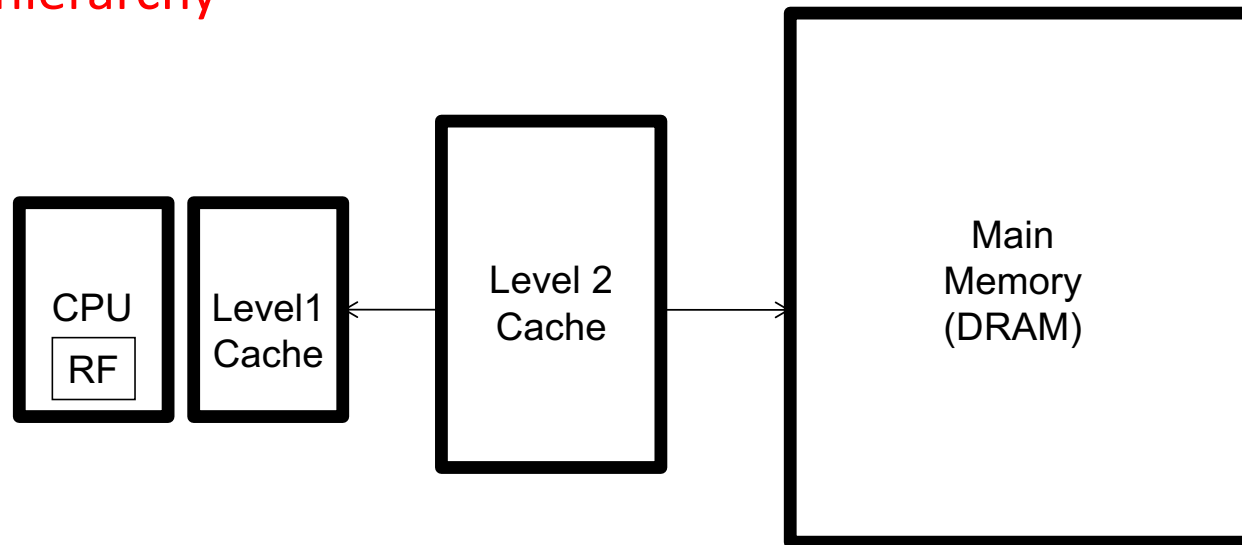


# The Bookshelf Analogy

- Book in your hand
- Desk
- Bookshelf
- Boxes at home
- Boxes in storage
  
- Recently-used books tend to stay on desk
  - Comp Arch books, books for classes you are currently taking
    - Until the desk gets full
- Adjacent books in the shelf needed around the same time
  - If I have organized/categorized my books well in the shelf

# Caching in a Pipelined Design

- The cache needs to be tightly integrated into the pipeline
  - Ideally, access in 1-cycle so that dependent operations do not stall
- High frequency pipeline → Cannot make the cache large
  - But, we want a large cache AND a pipelined design
- Idea: **Cache hierarchy**



## A Note on Manual vs. Automatic Management

- **Manual:** Programmer manages data movement across levels
  - too painful for programmers on substantial programs
  - still done in some embedded processors (on-chip scratch pad SRAM in lieu of a cache)
- **Automatic:** Hardware manages data movement across levels, **transparently to the programmer**
  - ++ programmer's life is easier
  - the average programmer doesn't need to know about it
    - You don't need to know how big the cache is and how it works to write a "correct" program! (What if you want a "fast" program?)

# Automatic Management in Memory Hierarchy

- Wilkes, “**Slave Memories and Dynamic Storage Allocation**,”  
IEEE Trans. On Electronic Computers, 1965.

## **Slave Memories and Dynamic Storage Allocation**

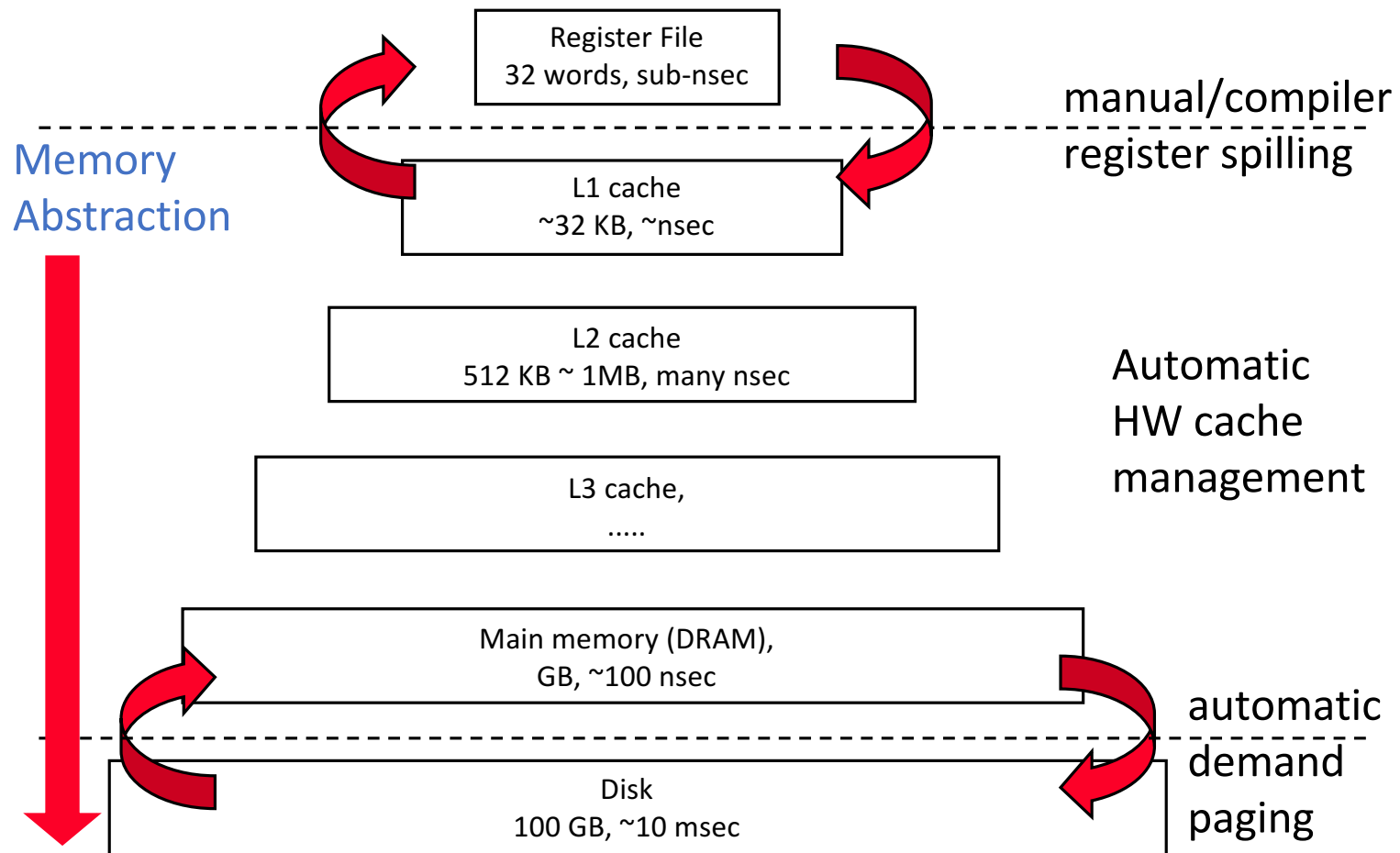
M. V. WILKES

### SUMMARY

The use is discussed of a fast core memory of, say, 32 000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.

- “By a slave memory I mean one which **automatically accumulates to itself words** that come from a slower main memory, and keeps them available for subsequent use without it being necessary for the penalty of main memory access to be incurred again.”

# A Modern Memory Hierarchy





# Hierarchical Latency Analysis

- For a given memory hierarchy level  $i$  it has a technology-intrinsic access time of  $t_i$ . The perceived access time  $T_i$  is longer than  $t_i$
- Except for the outer-most hierarchy, when looking for a given address there is
  - a chance (hit-rate  $h_i$ ) you “hit” and access time is  $t_i$
  - a chance (miss-rate  $m_i$ ) you “miss” and access time  $t_i + T_{i+1}$
  - $h_i + m_i = 1$

- Thus

$$T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$$

$$T_i = t_i + m_i \cdot T_{i+1}$$

- Miss-rate of just the references that missed at  $L_{i-1}$

# Hierarchy Design Considerations

- Recursive latency equation

$$T_i = t_i + m_i \cdot T_{i+1}$$

- The goal: achieve desired  $T_1$  within allowed cost
- $T_i \approx t_i$  is desirable
- Keep  $m_i$  low
  - increasing capacity  $C_i$  lowers  $m_i$ , but beware of increasing  $t_i$
  - lower  $m_i$  by smarter management (replacement::anticipate what you don't need, prefetching::anticipate what you will need)
- Keep  $T_{i+1}$  low
  - faster lower hierarchies, but beware of increasing cost
  - introduce intermediate hierarchies as a compromise

# Intel Pentium 4 Example

- 90nm P4, 3.6 GHz
  - L1 D-cache
    - $C_1 = 16K$
    - $t_1 = 4 \text{ cyc int} / 9 \text{ cycle fp}$
  - L2 D-cache
    - $C_2 = 1024 \text{ KB}$
    - $t_2 = 18 \text{ cyc int} / 18 \text{ cyc fp}$
  - Main memory
    - $t_3 = \sim 50\text{ns or } 180 \text{ cyc}$
  - Notice
    - best case latency is not 1
    - worst case access latencies are into 500+ cycles
- if  $m_1=0.1, m_2=0.1$   
 $T_1=7.6, T_2=36$
- if  $m_1=0.01, m_2=0.01$   
 $T_1=4.2, T_2=19.8$
- if  $m_1=0.05, m_2=0.01$   
 $T_1=5.00, T_2=19.8$
- if  $m_1=0.01, m_2=0.50$   
 $T_1=5.08, T_2=108$