LUND UNIVERSITY

# EITF20: Computer Architecture

## Part 6.1.1: Course Summary

Liang Liu
liang.liu@eit.lth.se

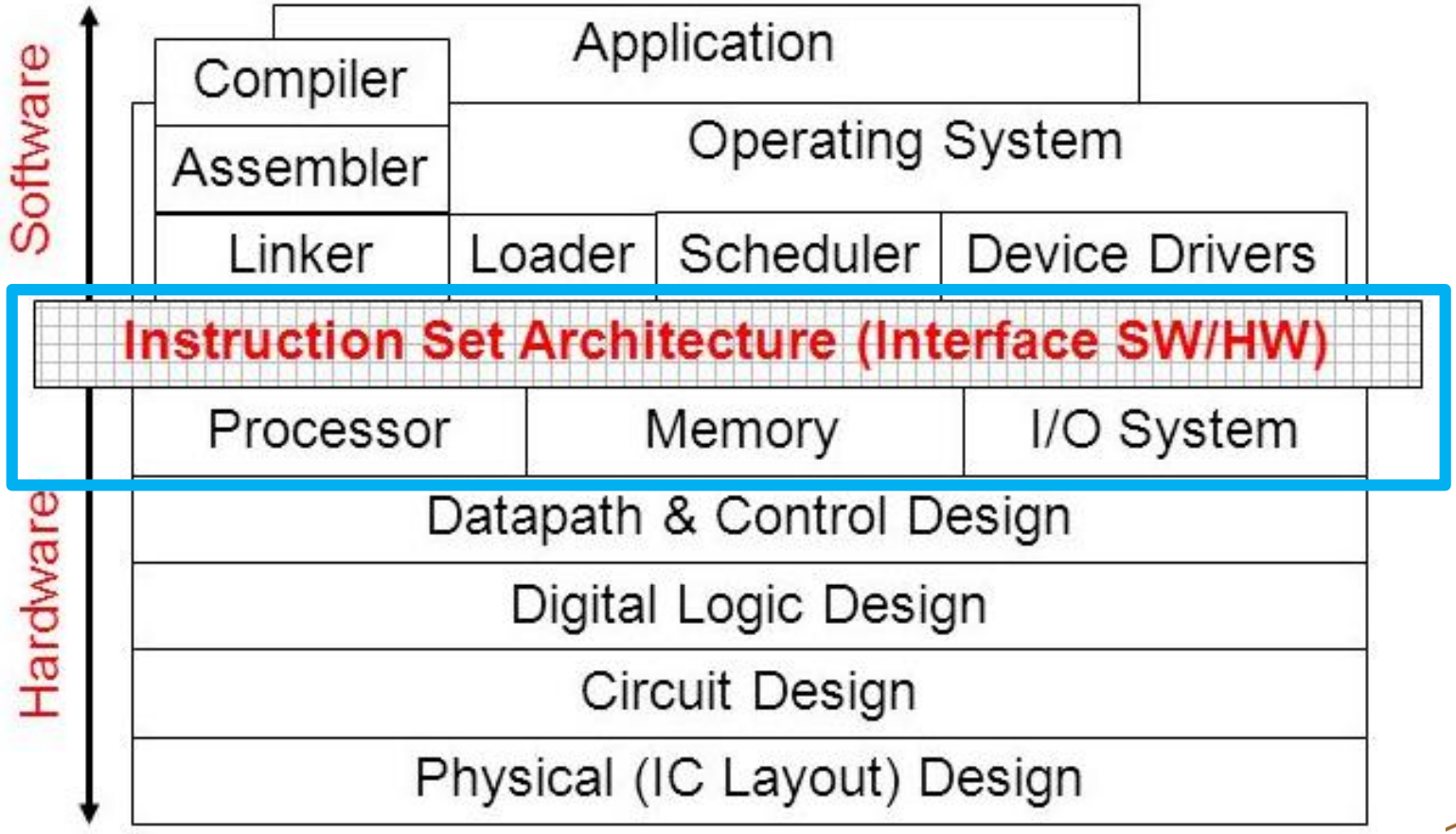
***The art of designing computers is based on engineering principles and quantitative performance evaluation***

# Computer Architecture

***Computer architecture is a set of disciplines that describe the functionality, organization and implementation of computer systems.***

- **ISA: Instruction-set architecture**
- **Computer orgnization: micro architecture**
- **Specific implementation**

# Computer abstraction levels



# What computer architecture?

- **Design and analysis**
  - ISA
  - Orgnization (microarchitecture)
  - Implementation
- **To meet requirements of**
  - Functionality (application, standards…)
  - Price
  - Performance
  - Power
  - Reliability
  - Dependability
  - Compatability
  - ..

# Outline

- Performance
- ISA
- Pipeline
- Memory Hierarchy
- I/O, MultiProcessor

# What is Performance?

| Plane | DC to Paris | Speed | Persons | Throughput (pkm/h) |
|---|---|---|---|---|
| Boeing 747 | 6.5 h | 980 km/h | 470 | 460 000 |
| Concorde | 3 h | 2160 km/h | 132 | 285 120 |

- **Time to complete a task ($T_{exe}$)**
  - Execution time, response time, latency
- **Task per day, hour…**
  - Total amount of tasks for given time
  - Thoughput, bandwidth
- **Speed of Concorde vs Boeing 747**
- **Throughput of Boeing 747 vs Concorde**

# Performance

$$Performance(X) = \frac{1}{T_{exe}(X)}$$

"X is n times faster than Y" means:

$$\frac{T_{exe}(Y)}{T_{exe}(X)} = \frac{Performance(X)}{Performance(Y)} = n$$

# Aspect of CPU performance

$$\text{CPUtime} = \text{Execution time} = \textit{seconds/program} =$$

$$\underbrace{(\textit{executed})\textit{instr./program}}_{IC} * \underbrace{\textit{cycles/instr.}}_{CPI} * \underbrace{\textit{seconds/cycle}}_{T_c}$$

| | IC | CPI | $T_c$ |
|---|---|---|---|
| Program | X | | |
| Compiler | X | (X) | |
| Instr. Set | X | X | |
| Organization | | X | X |
| Technology | | | X |

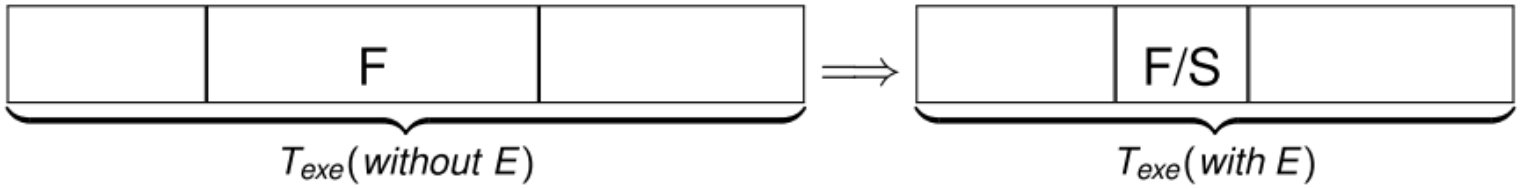
# Quantitative Principles

## This is intro to design and analysis

- Take advantage of parallelism
  - ILP, DLP, TLP, ...
- Principle of locality
  - 90% of execution time in only 10% of the code
- Focus on the common case
  - In makeing a design trade-off, favor the frequent case ove the infrequent case
- Amdahl's Law
  - The performance improvement gained from uisng faster mode is limited by the fraction of the time the faster mode can be used

# Amdahl's Law

Enhancement E accelerates a fraction F of a program by a factor S




Speedup due to enhancement E:

$$Speedup(E) = \frac{T_{exe}(without\ E)}{T_{exe}(with\ E)} = \frac{Performance(with\ E)}{Performance(without\ E)}$$

$$T_{exe}(with\ E) = T_{exe}(without\ E) * [(1 - F) + F/S]$$

$$Speedup(E) = \frac{T_{exe}(without\ E)}{T_{exe}(with\ E)} = \frac{1}{(1-F)+F/S}$$

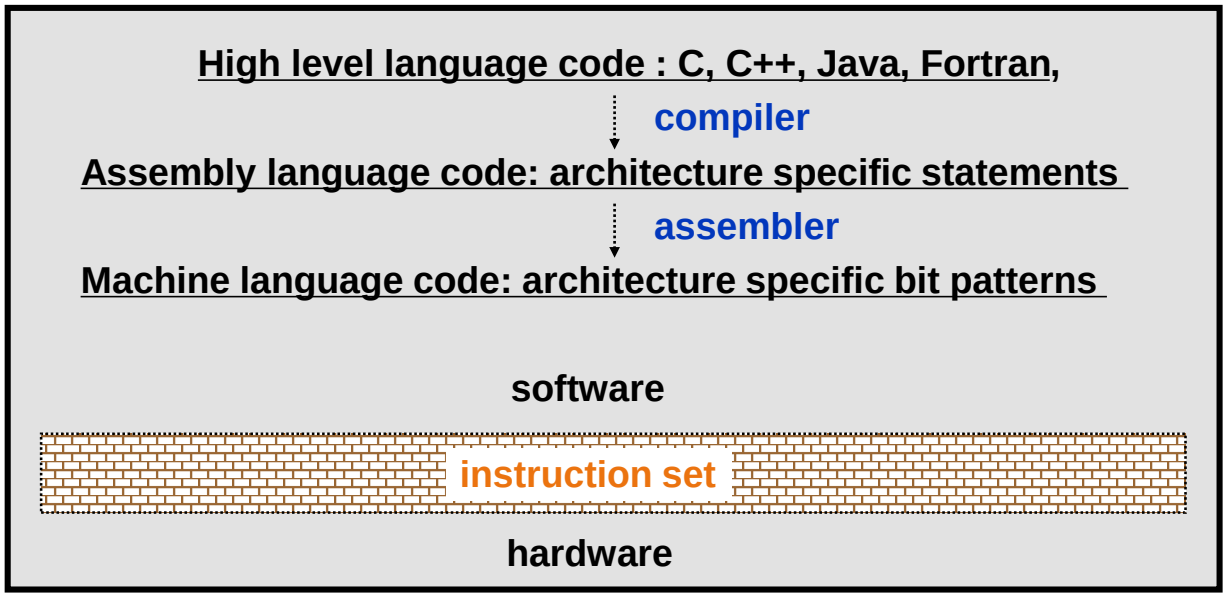
# Outline

- Performance
- **ISA**
- Pipeline
- Memory Hierarchy
- I/O, Storage System

# Interface Design

## A good interface

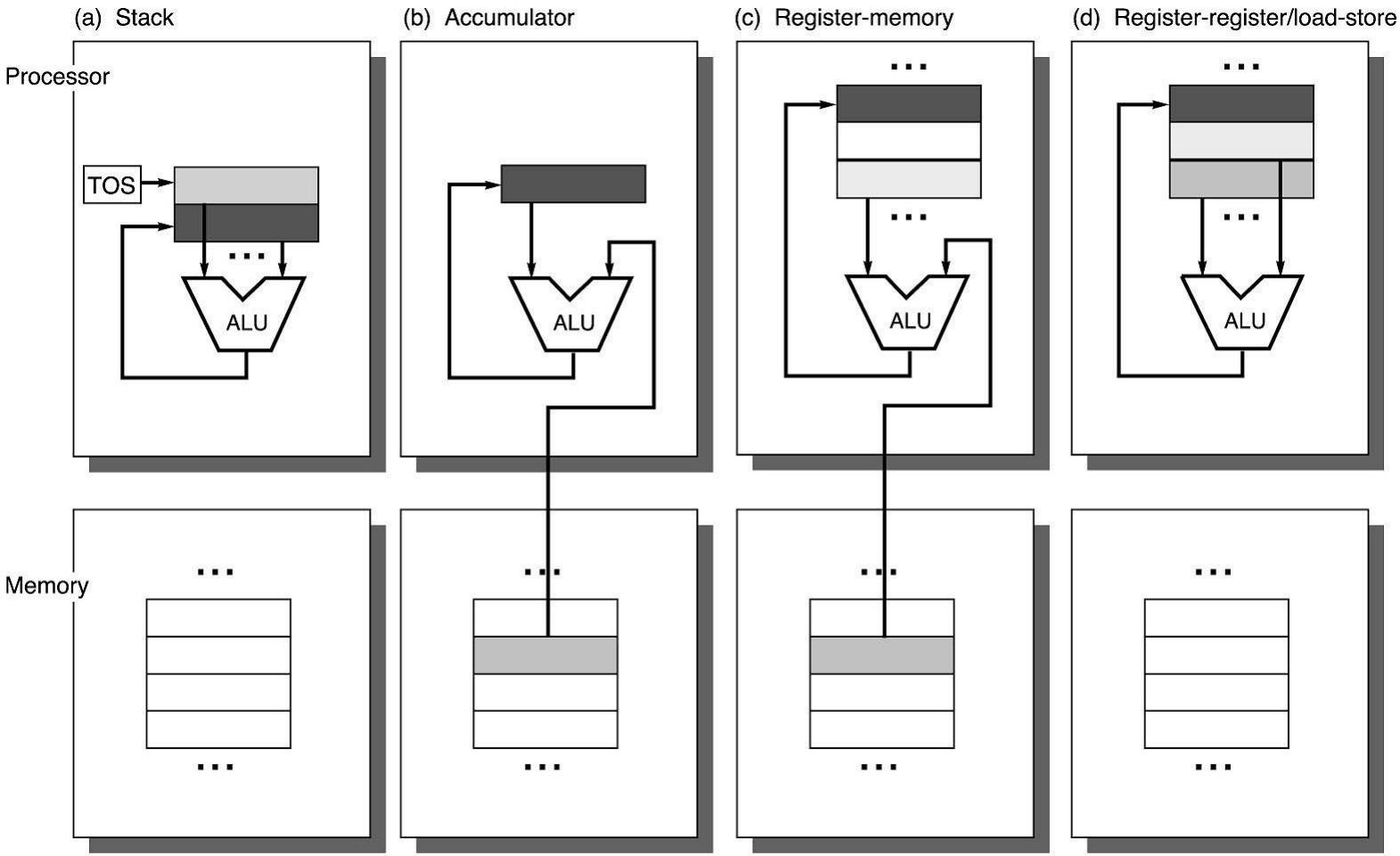
- Lasts through many implementations (portability, compatibility)
- Can be used in many different ways (generality)
- Provides sufficient functionality to higher levels
- Permits an efficient implementation at lower levels




# ISA Classification

- **What's needed in an instruction set?**
  - Addressing
  - Operands
  - Operations
  - Control Flow
- **Classification of instruction sets**
  - Register model
  - The number of operands for instructions
  - Addressing modes
  - The operations provided in the instruction set
  - Type and size of operands
  - Control flow instructions
  - Encoding

# ISA Classes: Where are operands stored



# Memory Addressing Mode

| Addressing Mode | Example | Action |
|---|---|---|
| 1. Register direct | Add R4, R3 | R4 <- R4 + R3 |
| 2. Immediate | Add R4, #3 | R4 <- R4 + 3 |
| 3. Displacement | Add R4, 100(R1) | R4 <- R4 + M[100 + R1] |
| 4. Register indirect | Add R4, (R1) | R4 <- R4 + M[R1] |
| 5. Indexed | Add R4, (R1 + R2) | R4 <- R4 + M[R1 + R2] |
| 6. Direct | Add R4, (1000) | R4 <- R4 + M[1000] |
| 7. Memory Indirect | Add R4, @(R3) | R4 <- R4 + M[M[R3]] |
| 8. Auto-increment | Add R4, (R2)+ | R4 <- R4 + M[R2]<br>R2 <- R2 + d |
| 9. Auto-decrement | Add R4, (R2)- | R4 <- R4 + M[R2]<br>R2 <- R2 - d |
| 10. Scaled | Add R4, 100(R2)[R3] | R4 <- R4 +<br>M[100 + R2 + R3*d] |

# Instruction format

## Variable instruction format

- Compact code but the instruction decoding is more complex and thus slower
- Examples: VAX, Intel 80x86 (1-17 byte)

| Operation # operands | Address specifier 1 | Address field 1 | ... | Address specifier x | Address field x |
|---|---|---|---|---|---|

## Fixed instruction format

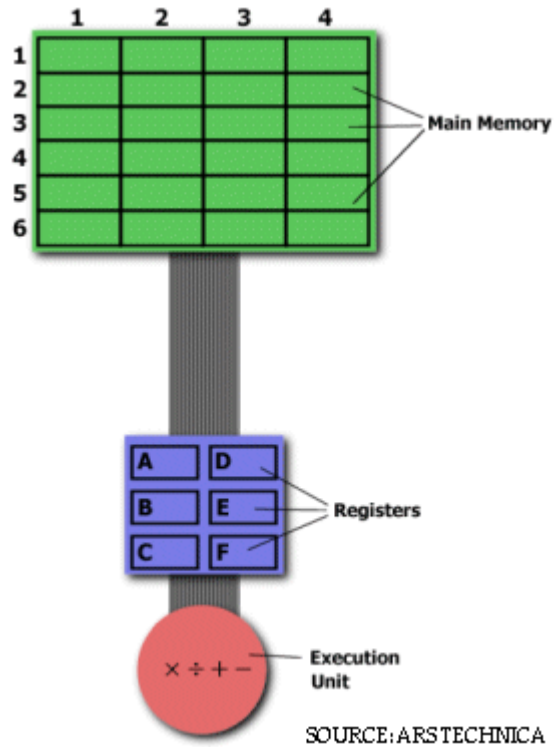
- Easy and fast to decode but gives large code size
- Examples: Alpha, ARM, MIPS (4byte), PowerPC, SPARC

| Operation | Address field 1 | Address field 2 | Address field 3 |
|---|---|---|---|

# Example: RISC-CICS

```
MULT 2:3, 5:2
```

```
LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A
```




| CISC | RISC |
|---|---|
| Emphasis on hardware | Emphasis on software |
| Includes multi-clock complex instructions | “Single”-clock, reduced instruction only |
| Memory-to-memory: "LOAD" and "STORE" incorporated in instructions | Register to register: "LOAD" and "STORE" are independent instructions |
| Small code sizes, high cycles per second | Low cycles per second, large code sizes |
| Irregular Instruction size | Regular Instruction size |

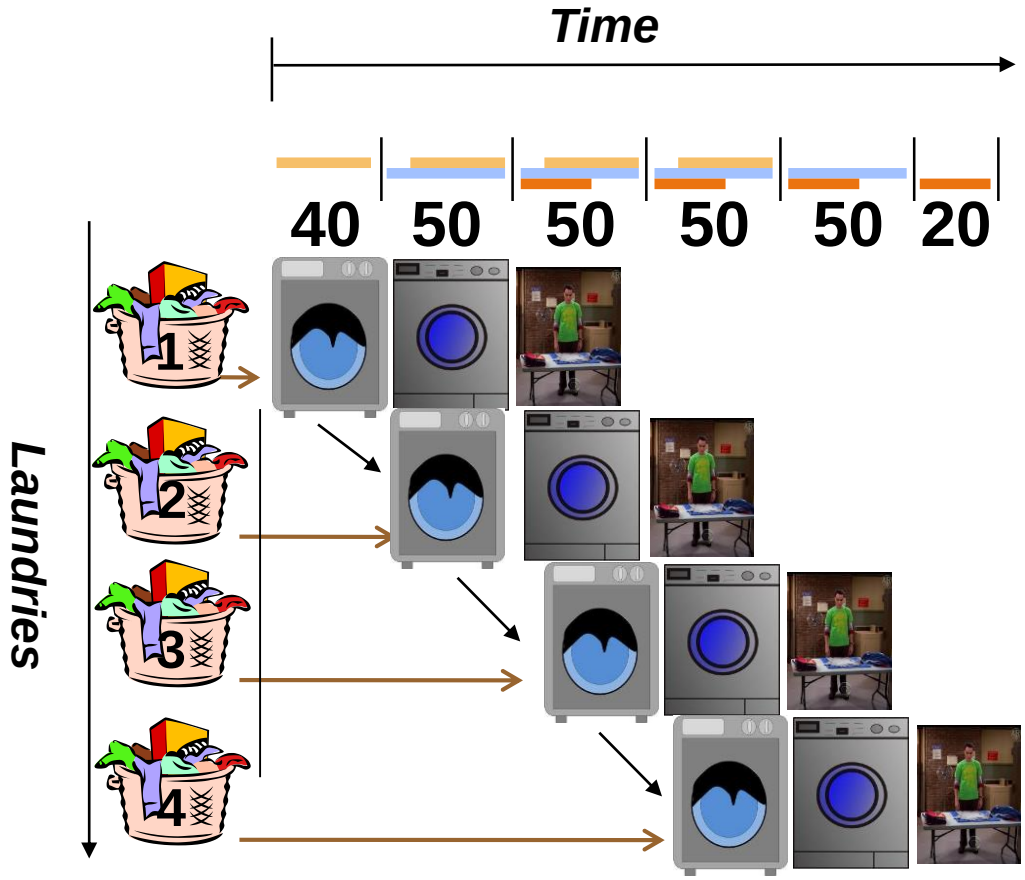
# Example: RISC-CICS

- **RISC (Reduced Instruction Set Computing)**
  - Simple instructions
  - MIPS, ARM, ...
  - Easier to design, build
  - Less power, in normal voltage supply
  - Larger code size (IC)
  - Easier for compiler
- **CISC (Complex Instruction Set Computing)**
  - Complex instructions
  - VAX, Intel 80x86 (now RISC-like internally), ...

# Outline

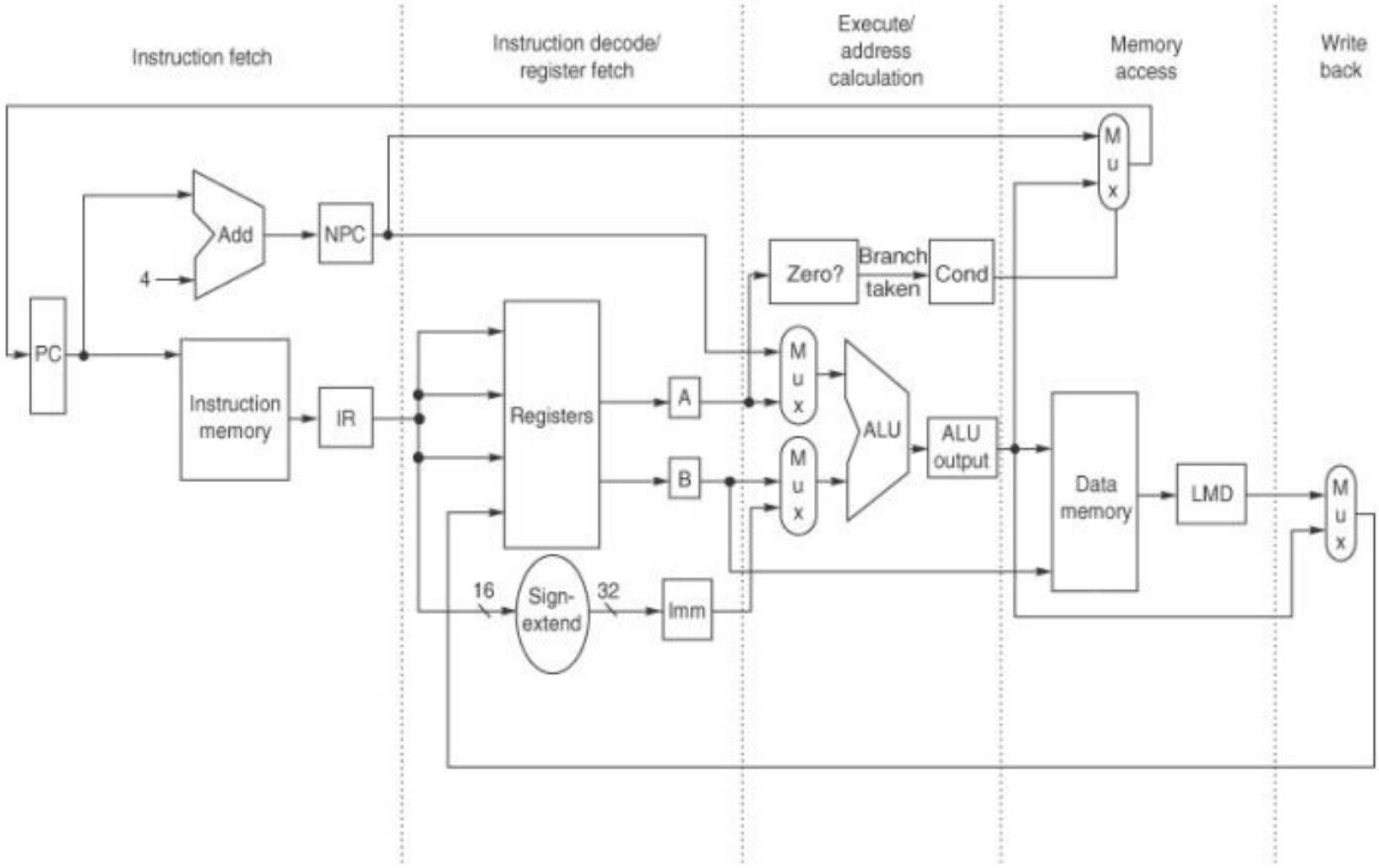
- Performance
- ISA
- **Pipeline**
- Memory Hierarchy
- I/O, Storage System

# Pipeline Facts




- Multiple tasks operating simultaneously
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Potential speedup ∝ Number of pipe stages

# One core – the MIPS data-path



# Dependencies

- **Data dependent: if**
  - Instruction *i* produces a result used by instr. *j*, or
  - Instruction *j* is data dependent on instruction k and instr. k is data dependent on instr. *i*

```
            LD      F0,0(R1)
Example:    ADDD    F4,F0,F2
            SD      0(R1),F4
```

- **Name dependent: two instructions use same name (register or memory address) but do not exchange data**
  - Anti-dependence (WAR if hazard in HW)

```
ADDD    F2,F0,F2    ; Must execute before LD
LD      F0,0(R1)
```

  - Output dependence (WAW if hazard in HW)

```
ADDD    F0,F2,F2    ; Must execute before LD
LD      F0,0(R1)
```

# Control dependencies

- **Determines order between an instruction and a branch instruction**

Example:

**if Test1 then { S1 }**

**if Test2 then { S2 }**

---

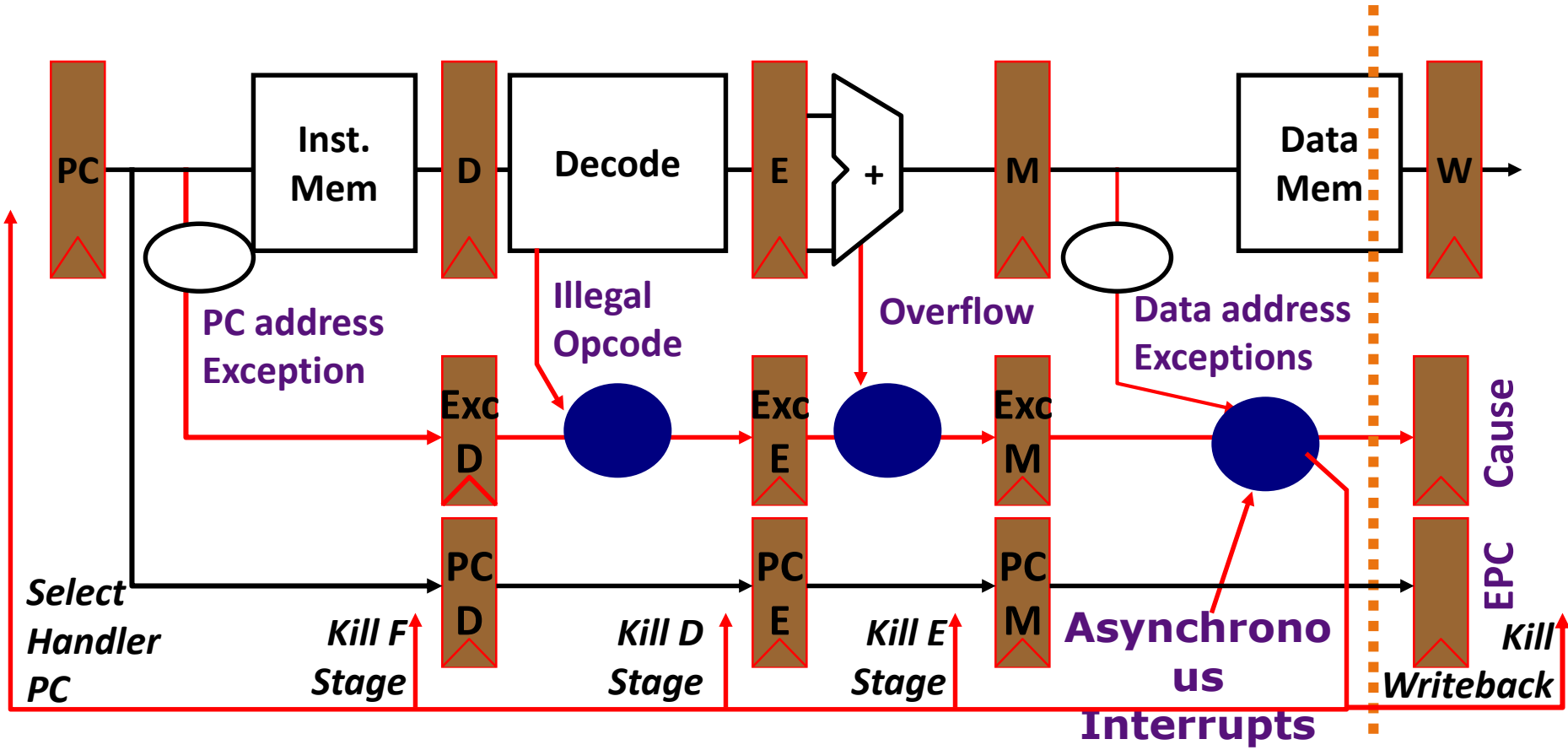
S1 is control dependent on Test1

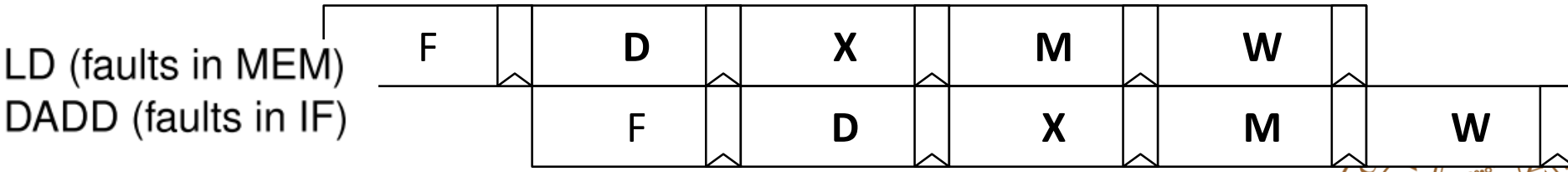
S2 is control dependent on Test2; but *not* on Test1

# Summary pipeline - method

| Dependency | Hazard | Method |
|---|---|---|
| Data | RAW | Forwarding, Scheduling, |
| Name | WAR, WAW | Register Renaming |
| Control | Control | Branch Prediction, Speculation, Delayed branch |
| Precise exceptions | | in-order commit |
| ILP | | Scheduling, Loop unrolling |

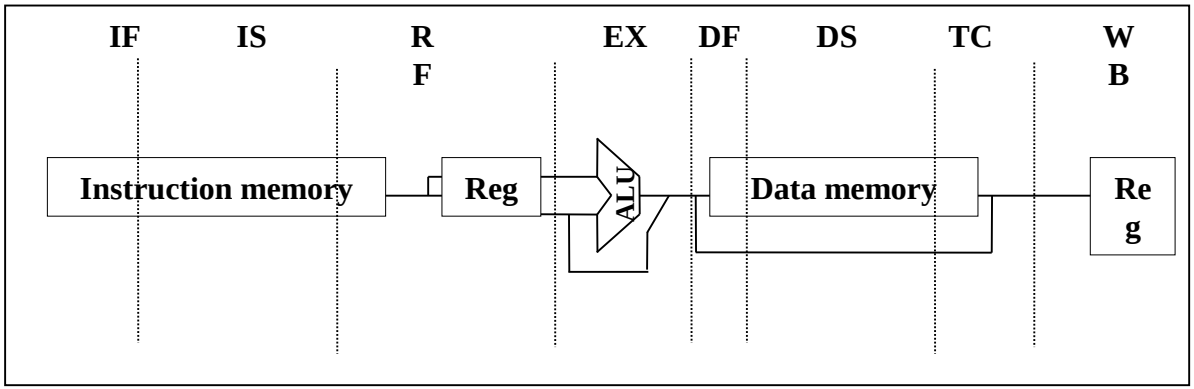
# Exception: solution for simple MIPS



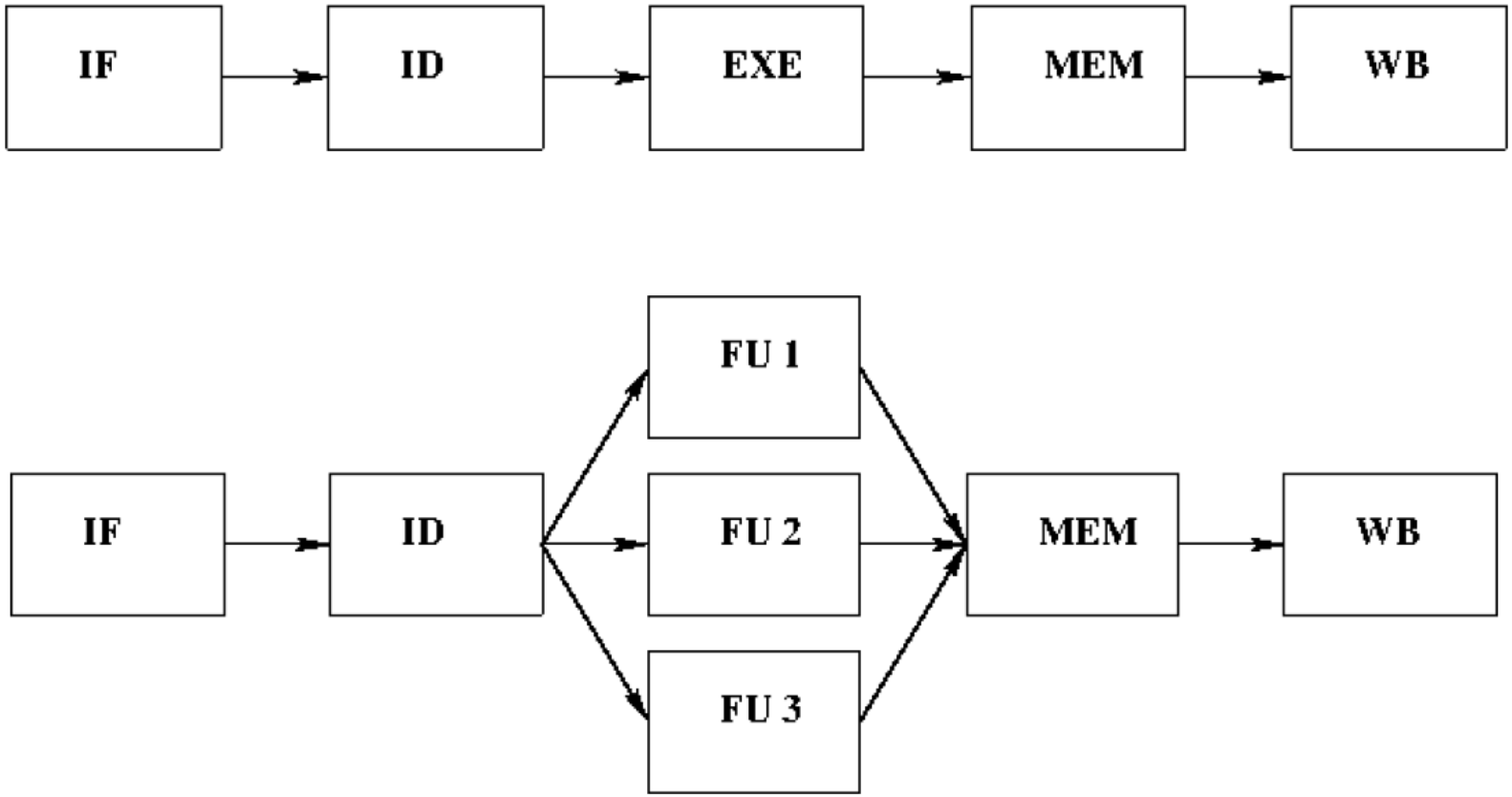



# Deeper pipeline

- **Implications of deeper pipeline**
  - Load latency: 2 cycles
  - Branch latency: 3 cycles (incl. one delay slot)
  - Bypassing (forwarding) from more stages
  - More instructions “in flight” in pipeline
  - Faster clock, larger latencies, more stalls
- **Performance equation: CPI ∗ Tc must be lower for the longer pipeline to make it worthwhile**



# Pipeline



# Pipeline hazard

- **RAW hazards:**
  - Normal bypassing from MEM and WB stages
  - Stall in ID stage if any of the source operands is destination operand in any of the FP functional units
- **WAR hazards?**
  - There are no WAR-hazards since the operands are read (in ID) before the EX-stages in the pipeline
- **WAW hazard**

```
DIV.D    F0,F2,F3    FP divide 24 cycles
...
SUB.D    F0,F8,F10   FP subtract 3 cycles
```

  - SUB finishes before DIV which will overwrite the result from SUB!
  - are eliminated by stalling SUB until DIV reaches MEM stage
  - When WAW hazard is a problem?

# Compiler optimization

```
loop:   LD      F0, 0(R1)       ; F0 = array element
        ADDD    F4, F0, F2      ; Add scalar constant
        SD      F4, 0(R1)       ; Save result
        DADDUI  R1, R1, #-8     ; decrement array ptr.
        BNE     R1,R2, loop     ; reiterate if R1 != R2
```

## Loop unrolling

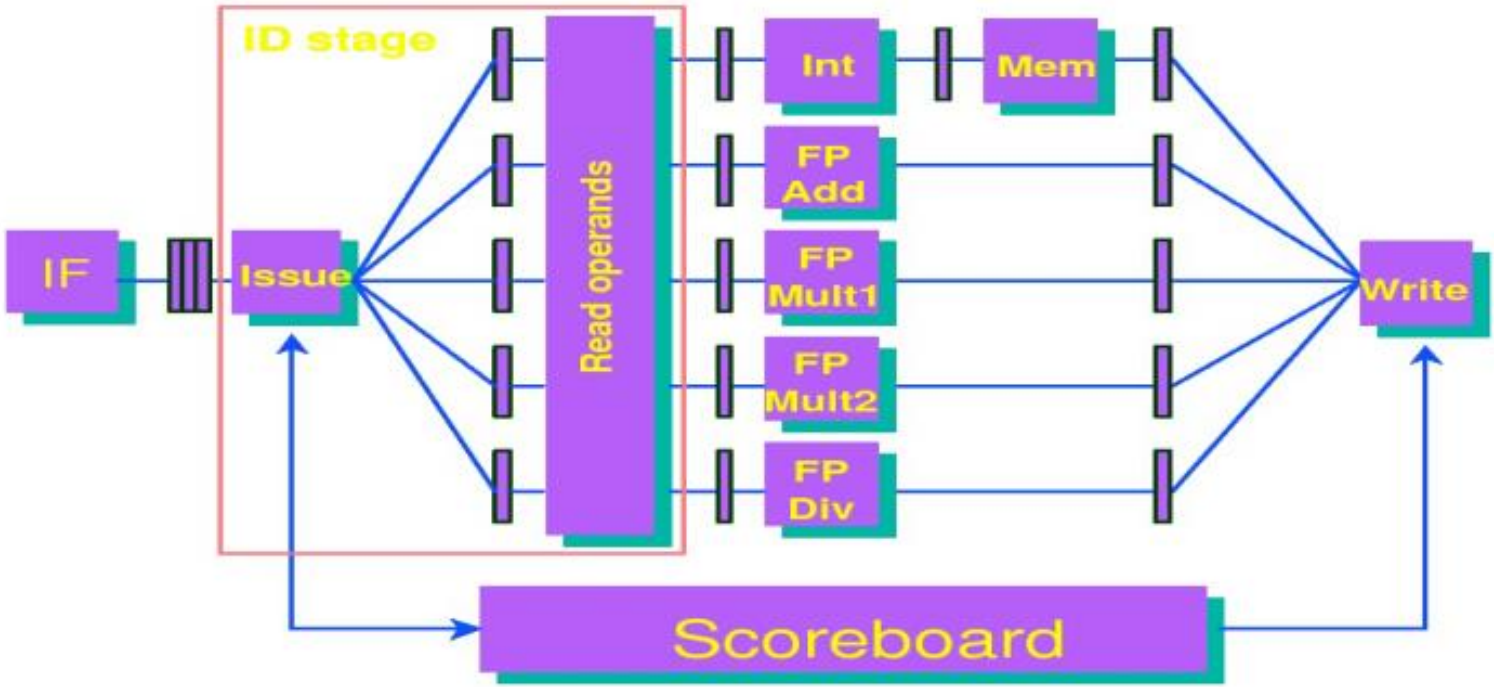
```
1   loop:   LD      F0, 0(R1)
2           ADDD    F4, F0, F2
3           SD      F4, 0(R1)
4           LD      F6, -8(R1)
5           ADDD    F8, F6, F2
6           SD      F8, -8(R1)
7           LD      F10, -16(R1)
8           ADDD    F12, F10, F2
9           SD      F12, -16(R1)
10          LD      F14, -24(R1)
11          ADDD    F16, F14, F2
12          SD      F16, -24(R1)
13          DADDUI  R1, R1, #-32
14          BNE     R1, R2, loop
```

## Scheduling

```
Loop:   L.D     F0,0(R1)
        L.D     F6,-8(R1)
        L.D     F10,-16(R1)
        L.D     F14,-24(R1)
        ADD.D   F4,F0,F2
        ADD.D   F8,F6,F2
        ADD.D   F12,F10,F2
        ADD.D   F16,F14,F2
        S.D     F4,0(R1)
        S.D     F8,-8(R1)
        DADDUI  R1,R1,#-32
        S.D     F12,16(R1)
        S.D     F16,8(R1)
        BNE     R1,R2,Loop
```

# Scoreboard pipeline

- **Issue:** decode and check for structural & WAW hazards
- **Read operands:** wait until no data hazards, then read operands
- **All data hazards are handled by the scoreboard**




# Scoreboard functionality

- **Issue:** An instruction is issued if:
  - The needed functional unit is free (there is no **structural hazard**)
  - No functional unit has a destination operand equal to the destination of the instruction (resolves **WAW hazards**)
- **Read:** Wait until no data hazards, then read operands
  - Performed in parallel for all functional units
  - Resolves **RAW hazards** dynamically
- **EX:** Normal execution
  - Notify the scoreboard when ready
- **Write:** The instruction can update destination if:
  - All earlier instructions have read their operands (resolves **WAR hazards**)

# Scoreboard example

**Instruction status**

| Instruction | | j | k | Issue | Read ops | Exec. compl. | Write result |
|---|---|---|---|---|---|---|---|
| LD | F6 | 34+ | R2 | | | | |
| LD | F2 | 45+ | R3 | | | | |
| MULTD | F0 | F2 | F4 | | | | |
| SUBD | F8 | F6 | F2 | | | | |
| DIVD | F10 | F0 | F6 | | | | |
| ADDD | F6 | F8 | F2 | | | | |

**Functional unit status**

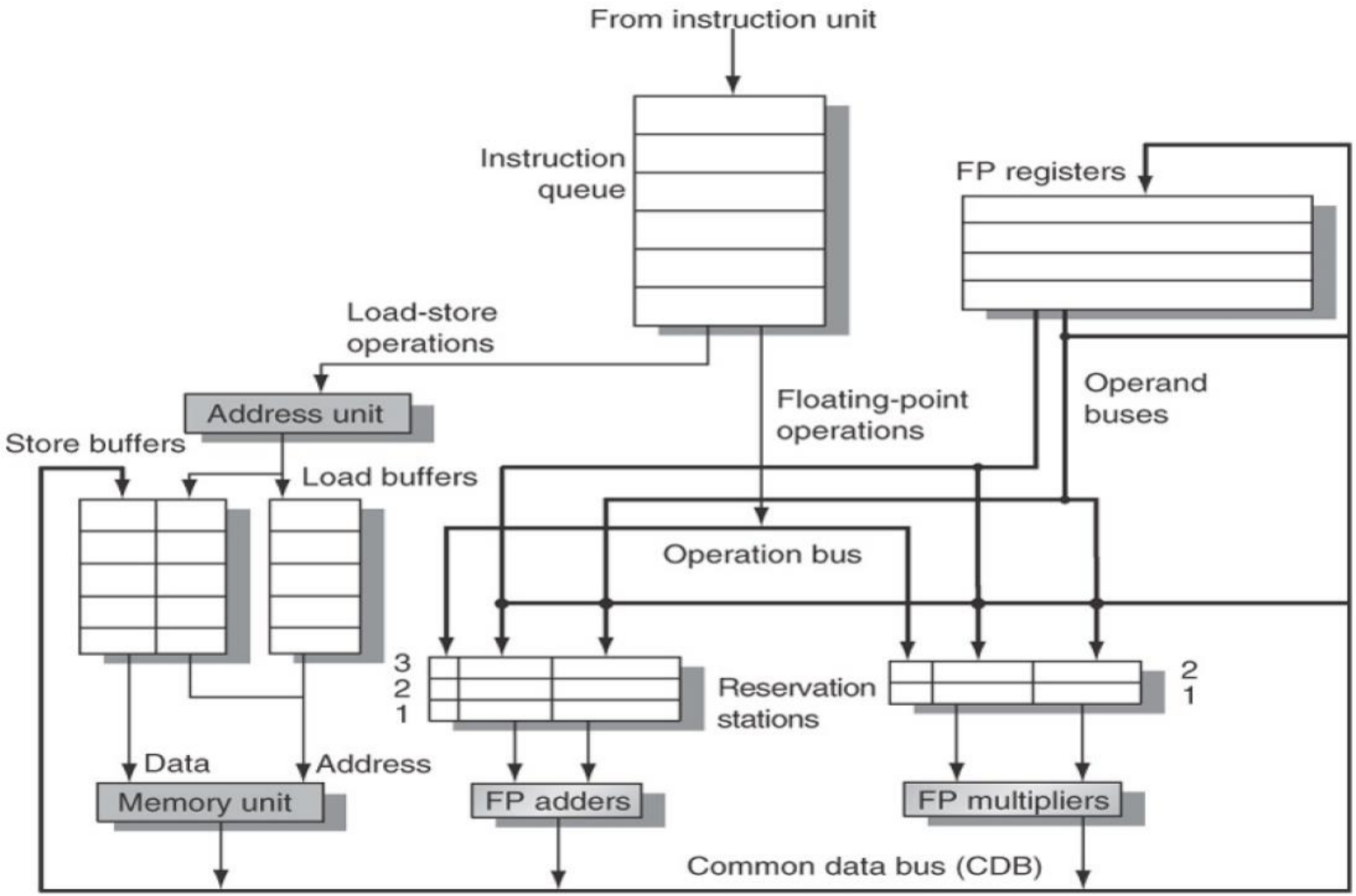
| Time | Name | Busy | Op | dest Fi | src 1 Fj | src 2 Fk | FUsrc1 Qj | FUsrc2 Qk | Fj? Rj | Fk? Rk |
|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | No | | | | | | | | |
| | Mult1 | No | | | | | | | | |
| | Mult2 | No | | | | | | | | |
| | Add | No | | | | | | | | |
| | Divide | No | | | | | | | | |

**Register result status**

| | F0 | F2 | F4 | F6 | F8 | F10 | ... | F30 |
|---|---|---|---|---|---|---|---|---|
| FU | | | | | | | | |

Clock: 0

# Tomasulo orgnizations



# Reservation stations

- **Op:Operation to perform (e.g., + or –)**
- **Vj, Vk: Value (instead of reg specifier) of Source operands**
- **Qj, Qk: Reservation stations (instead of FU) producing source registers (value to be written)**
  - Note: Qj,Qk=0 => ready
  - V and Q filed are mutual exclusive
- **Busy: Indicates reservation station or FU is busy**
- **Register result status—Indicates which RS will write each register**
  - Blank when no pending instructions that will write that register

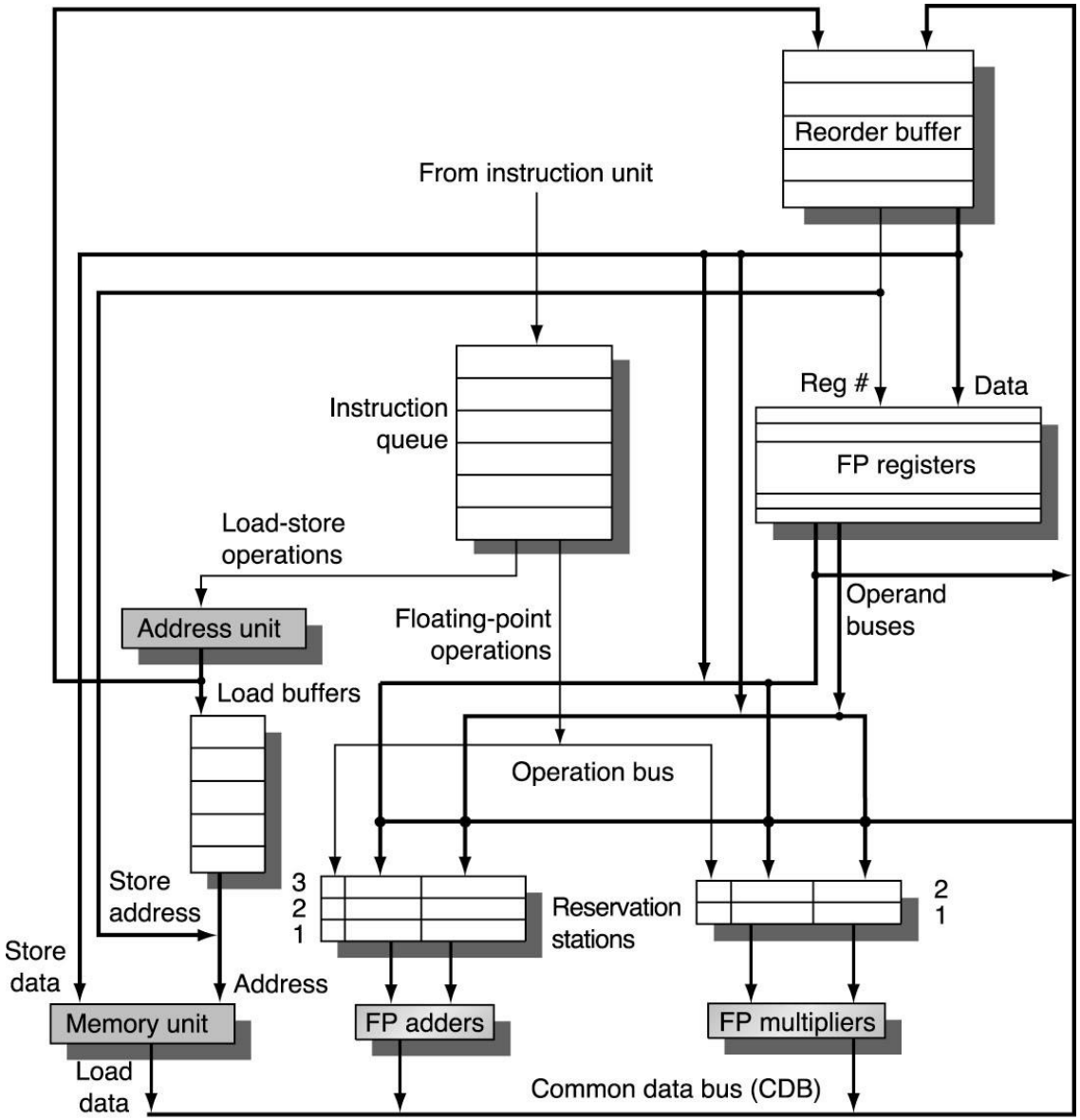
Functional unit status

| Time | Name | Busy | Op | src 1 Vj | src 2 Vk | RS for j Qj | RS for k Qk |
|---|---|---|---|---|---|---|---|
| | Add1 | No | | | | | |
| | Add2 | No | | | | | |
| | Add3 | No | | | | | |
| | Mult1 | No | | | | | |
| | Mult2 | No | | | | | |

# Three stages of Tomasulo algorithm

- **Issue –** get instruction from instruction Queue
  - If matching reservation station free (no **structural hazard**)
  - Instruction is issued together with its operands values or RS point (register rename, handle **WAR, WAW**)
- **Execution –** operate on operands (EX)
  - When both operands are ready, then execute (handle **RAW**)
  - If not ready, watch **Common Data Bus (CDB)** for operands (snooping)
- **Write result –** finish execution (WB)
  - Write on CDB to all awaiting RS, regs (**forwarding**)
  - Mark reservation station available

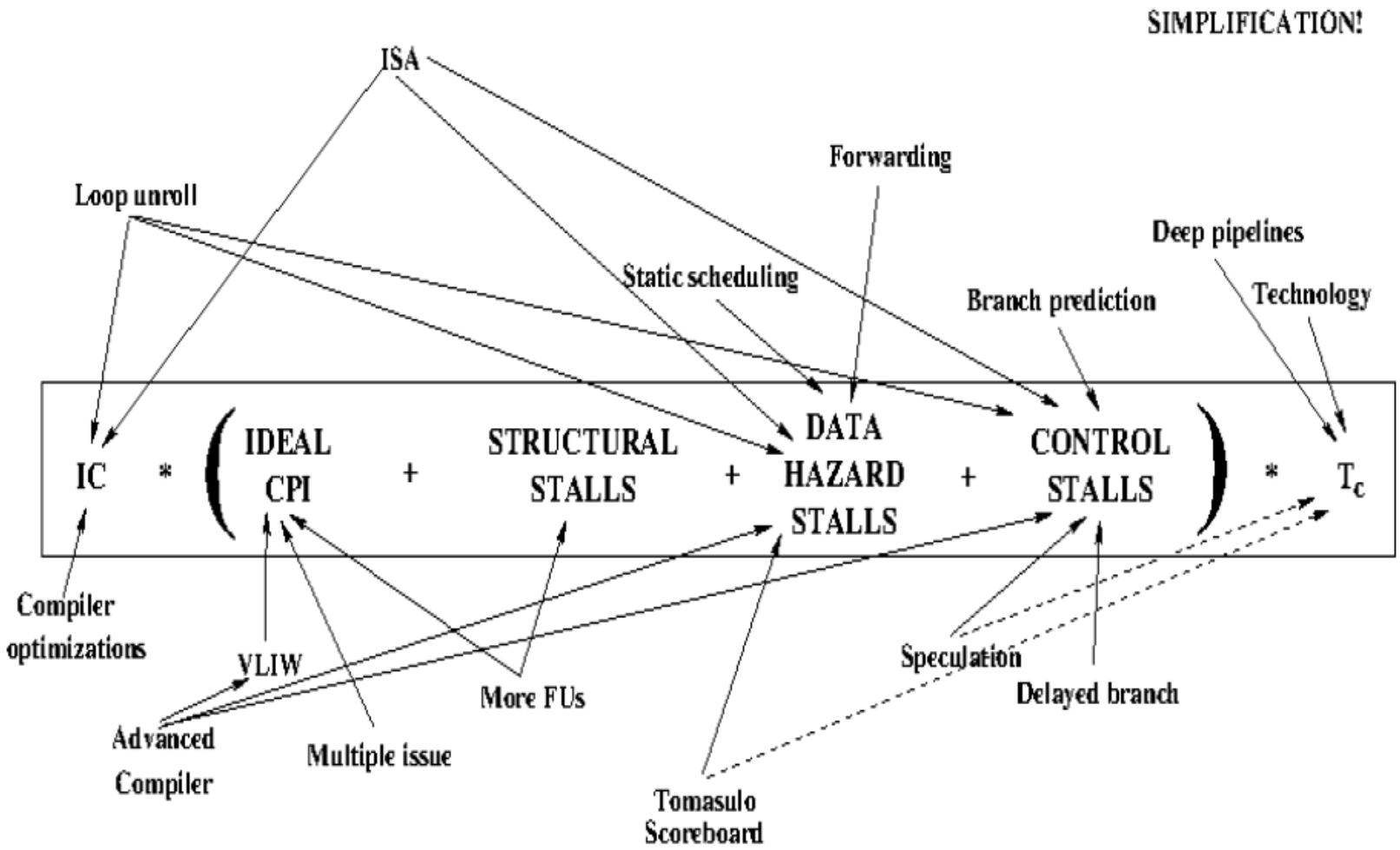
# Tomasulo extended to support speculation



# Summary pipeline - implementation

| Problem | Simple | Scoreboard | Tomasulo | Tomasulo + Speculation |
|---|---|---|---|---|
| | Static Sch | Dynamic Scheduling | | |
| RAW | forwarding stall | wait (Read) | CDB stall | CDB stall |
| WAR | - | wait (Write) | Reg. rename | Reg. rename |
| WAW | - | wait (Issue) | Reg. rename | Reg. rename |
| Exceptions | precise | ? | ? | precise, ROB |
| Issue | in-order | in-order | in-order | in-order |
| Execution | in-order | out-of-order | out-of-order | out-of-order |
| Completion | in-order | out-of-order | out-of-order | in-order |
| Structural hazard | - | many FU stall | many FU, CDB, stall | many FU, CDB, stall |
| Control hazard | Delayed br., stall | Branch prediction | Branch prediction | Br. pred, speculation |

# CPU performance equation




**+ Memory access**
**+ Communication**

# Outline

- Performance
- ISA
- Pipeline
- **Memory Hierarchy**
- I/O, Storage System

# Memory tricks (techniques)

Use a hierarchy

| CPU | superfast | Registers | | instructions |
|---|---|---|---|---|
| | FAST | Cache | | |
| | | | cache memory | (HW) |
| Memory | CHEAP | Main memory | | |
| | | | virtual memory | (SW) |
| | BIG | Disk | | |

# Levels of memory hierarchy

*Capacity*
*Access Time*
*Cost/bit*

*Staging*
*Transfer Unit*

Upper Level
*Faster*

*CPU Registers*
500 Bytes
0.25 ns
~$.01

Registers

Words

programmer/compiler
1-8 bytes

*Cache*
16K-1M Bytes
1 ns
~$10$^{-4}$

L1, L2, … Cache

Blocks

cache controller
8-128 bytes

*Main Memory*
64M-2G Bytes
100ns
~$10$^{-7}$

Memory

Pages

OS
4-64K bytes

*Disk*
100 G Bytes
5 ms
~$10$^{-7}$- 10$^{-9}$

Disk

Files

user/operator
Mbytes

*Tape/Network*
"infinite"
secs.
~$10$^{-10}$

Tape/Network

*Larger*
Lower Level

# Four memory hierarchy questions

- 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?

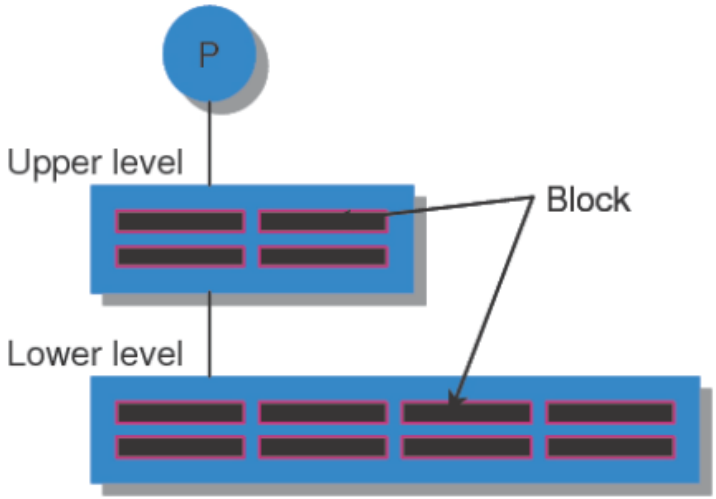
(Block identification)

- Q3: Which block should be replaced on a miss?

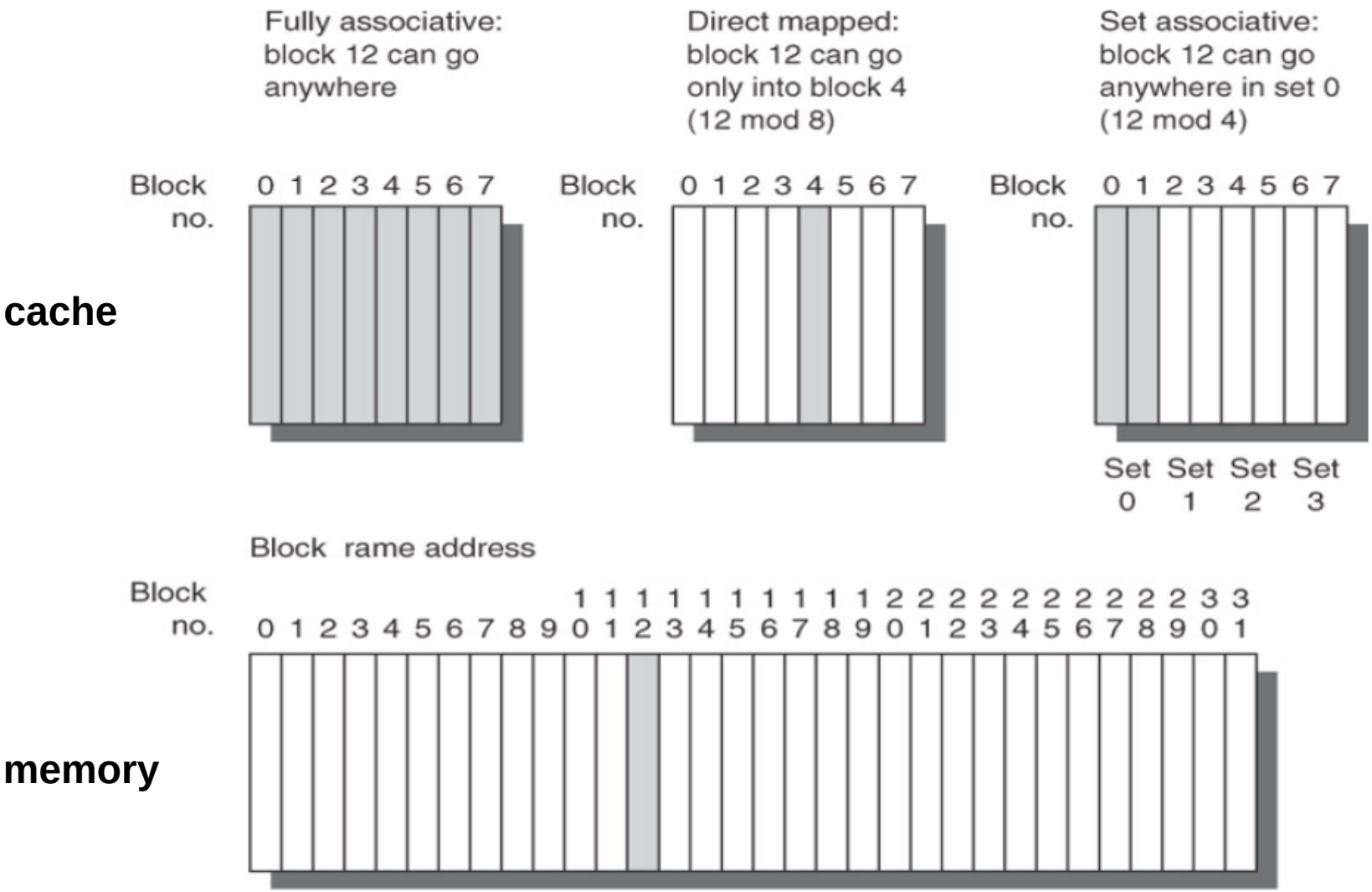
(Block replacement)

- Q4: What happens on a write?

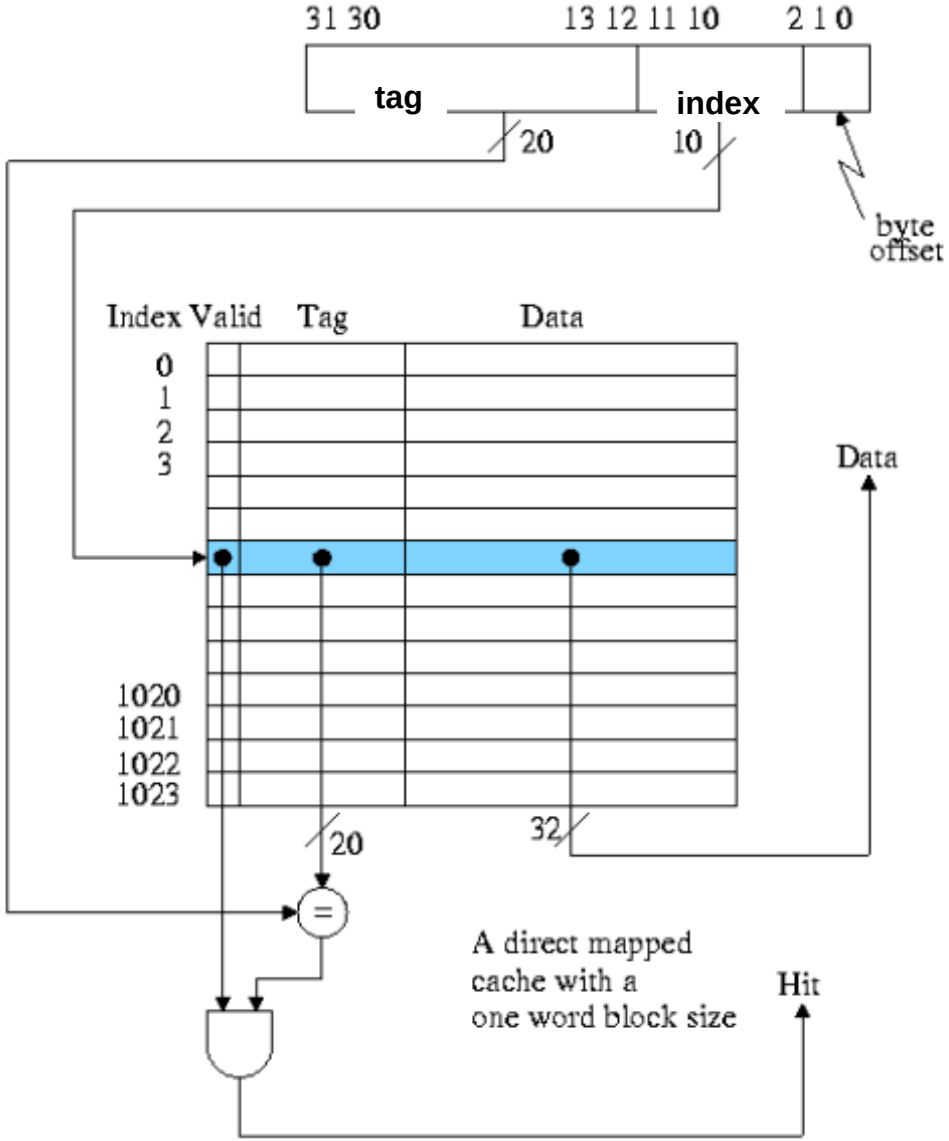
(Write strategy)




# Block placement



# Block identification



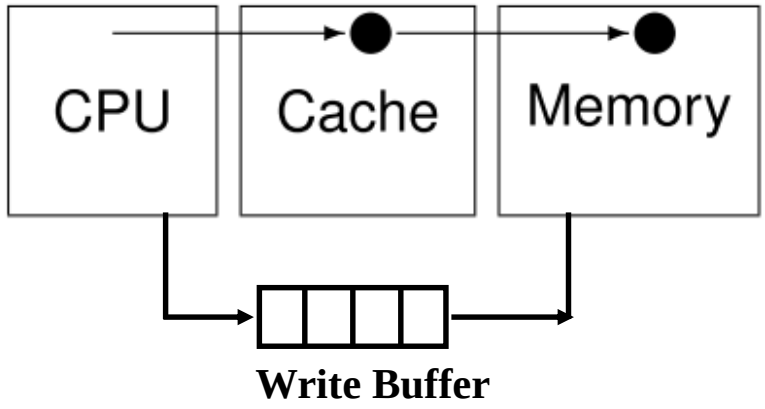
# Which block should be replaced on a Cache miss?

- **Direct mapped caches don't need a block replacement policy**
- **Primary strategies:**
  - Random (easiest to implement)
  - LRU – Least Recently Used (best, hard to implement)
  - FIFO – Oldest (used to approximate LRU)

# Cache write (hit)

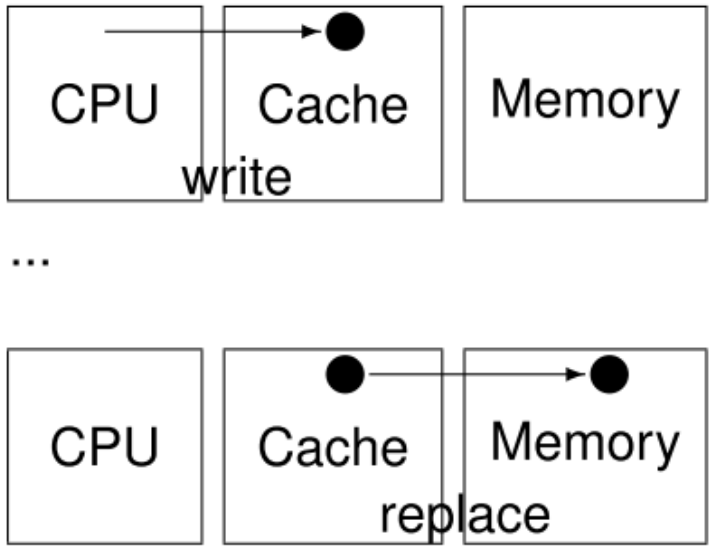
## Write through:

- The information is written to both the block in the cache and to the block in the lower-level memory
- Is always combined with write buffers so that the CPU doesn't have to wait for the lower level memory




## Write back:

- The information is written only to the block in the cache
- Copy a modified cache block to main memory only when replaced
- Is the block clean or modified? (dirty bit, several write to the same block)



# Cache performance

$$\text{Average memory access time} = \textit{hit time} + \textit{miss rate} * \textit{miss penalty}$$

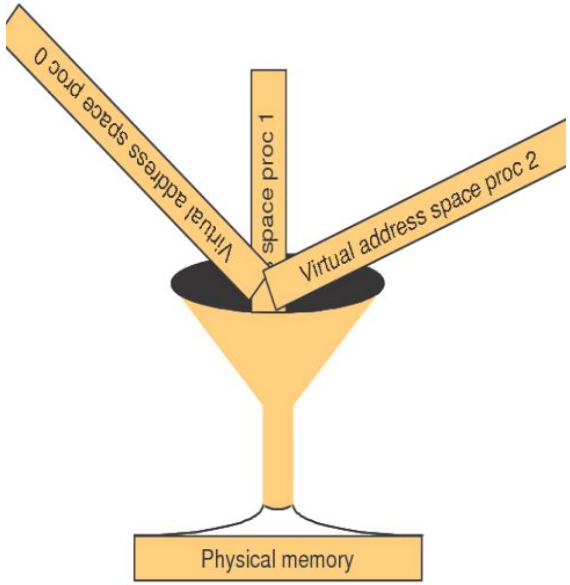
- **Three ways to increase performance:**
  - Reduce miss rate
  - Reduce miss penalty
  - Reduce hit time (improves $T_C$)

# Cache optimizations

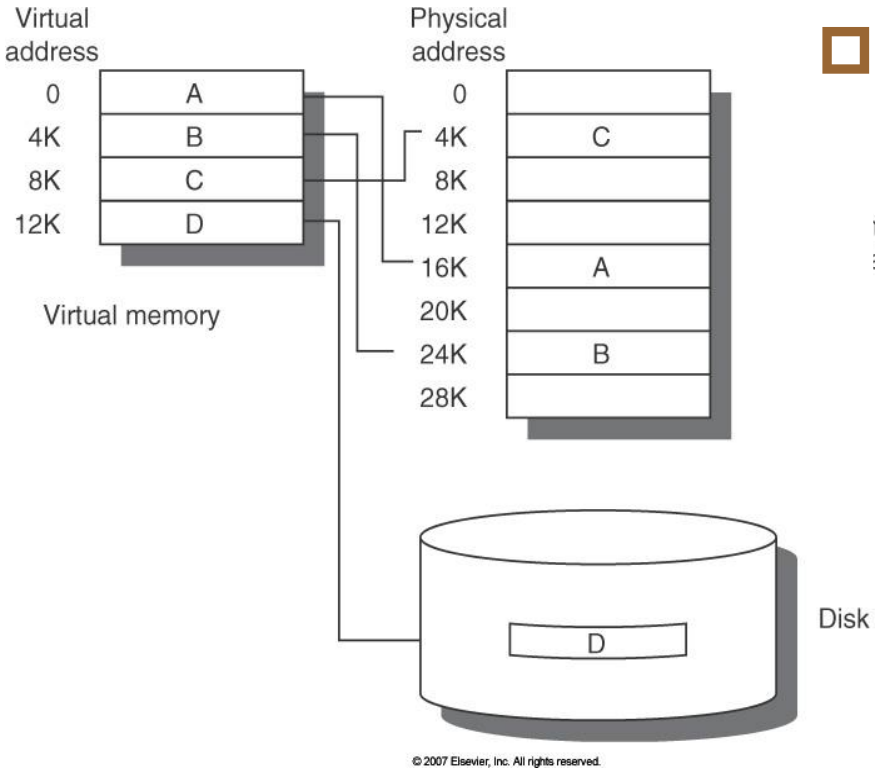
| | Hit time | Band-width | Miss penalty | Miss rate | HW complexity |
|---|---|---|---|---|---|
| Simple | + | | | - | 0 |
| Addr. transl. | + | | | | 1 |
| Way-predict | + | | | | 1 |
| Trace | + | | | | 3 |
| Pipelined | - | + | | | 1 |
| Banked | | + | | | 1 |
| Nonblocking | | + | + | | 3 |
| Early start | | | + | | 2 |
| Merging write | | | + | | 1 |
| Multilevel | | | + | | 2 |
| Read priority | | | + | | 1 |
| Prefetch | | | + | + | 2-3 |
| Victim | | | + | + | 2 |
| Compiler | | | | + | 0 |
| Larger block | | | - | + | 0 |
| Larger cache | - | | | + | 1 |
| Associativity | - | | | + | 1 |

# Virtual memory benifits

- **Using physical memory efficiently**
  - Allowing more than physical memory addressing
  - Enables programs to begin before loading fully
  - Programmers used to use overlays and manually control loading/unloading
- **Using physical memory simply**
  - Virtual memory simplifies memory management
  - Programmer can think in terms of a large, linear address space
- **Using physical memory safely**
  - Virtual memory protests process' address spaces
  - Processes cannot interfere with each other, because they operate in different address space
  - User processes cannot access priviledged information




# Virtual memory concept




- **Is part of memory hierarchy**
  - The virtual address space is divided into **pages** (blocks in Cache)
  - The physical address space is divided into **page frames**
  - A miss is called a **page fault**
  - Pages not in main memory are stored on **disk**

- **The CPU uses *virtual addresses***
- **We need an *address translation* (memory mapping) mechanism**

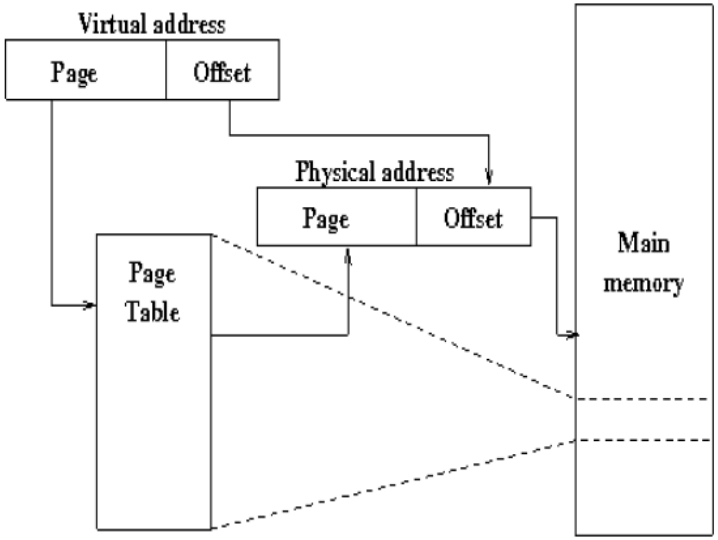
# Page placement

- **Where can a page be placed in main memory?**
  - Cache access: ~ ns
  - Memory access: ~ 100 ns
  - Disk access: ~ 10, 000, 000 ns

$\Longrightarrow$ **HIGH miss penalty**

- **The high miss penalty makes it**
  - Necessary to **minimize miss rate**
  - Possible to use software solutions to implement a **fully associative address** mapping

# Page identification: address mapping




- **Contains Real Page Number**
- **Miscellaneous control information**
  - valid bit,
  - dirty bit,
  - replacement information,
  - access control

- **4Byte per page table entry**
  - Page table will have

    $2^{20}*4=2^{22}=4MByte$
  - Generally stored in the main memory

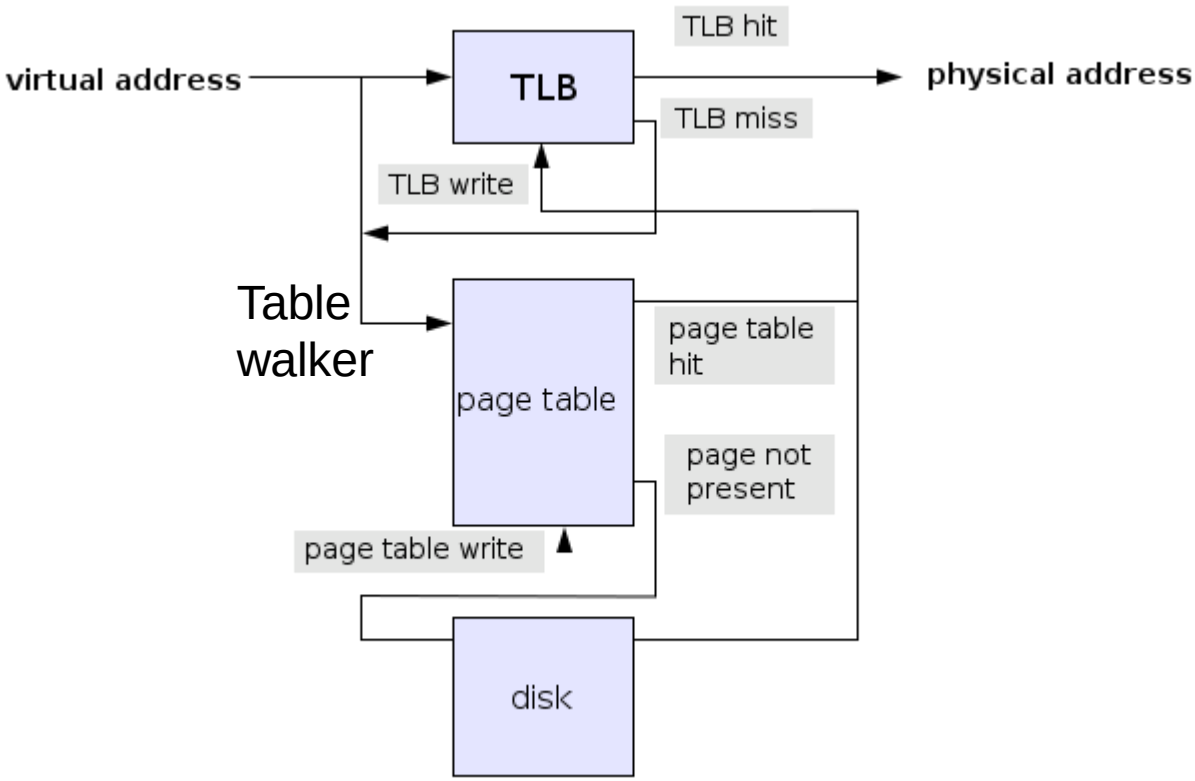
- **64 bit virtual address,16 KB pages:**

  $2^{64}/2^{14}*4=2^{52}=2^{12}TByte$

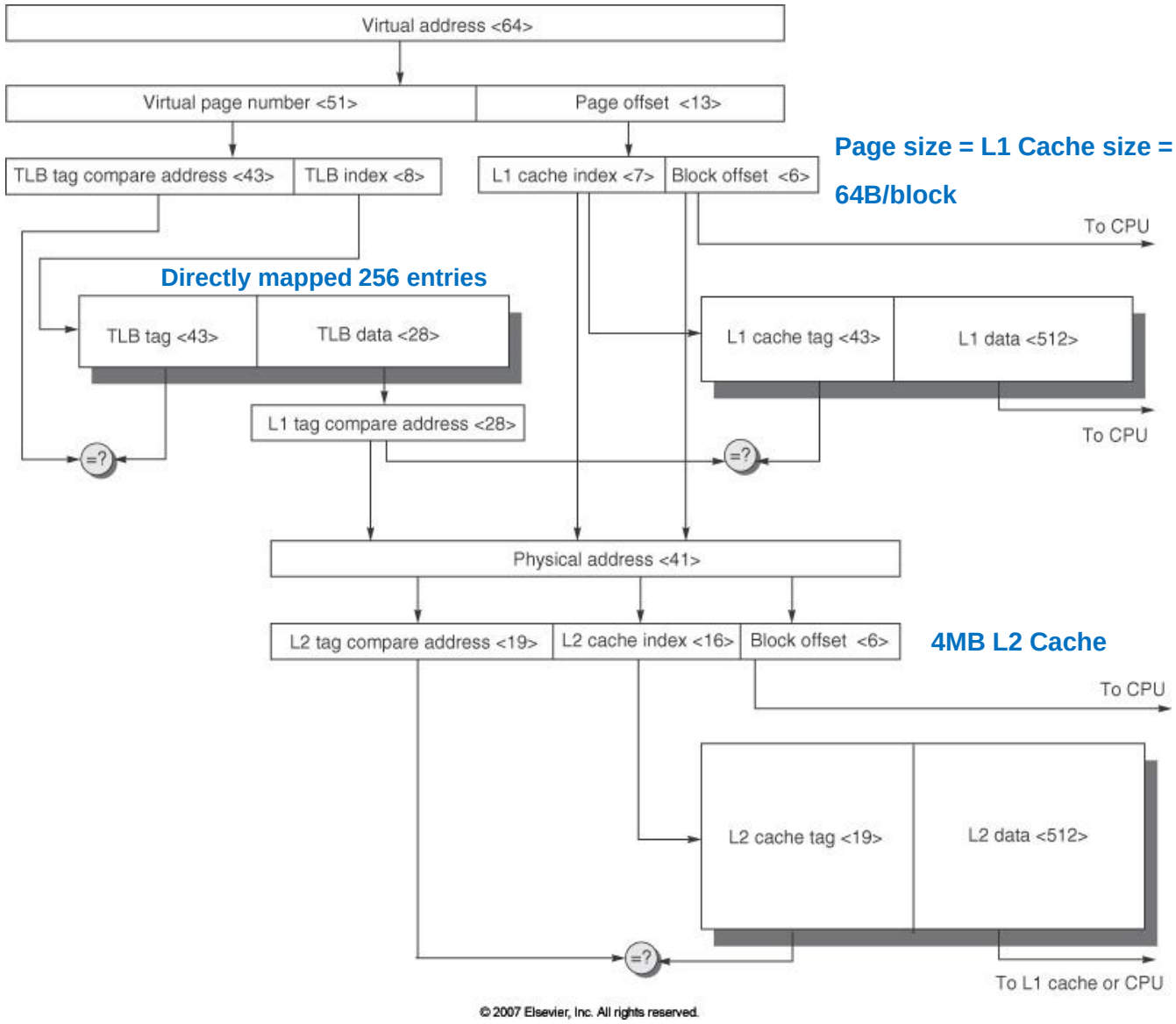
- **One page table per program (100 program?)**
- **Solutions**
  - Multi–level page table
  - Inverted page table

# Page identification (TLB)

- **How do we avoid two (or more) memory references for each original memory reference?**
  - Cache address translations – Translation Look-aside Buffer (TLB)




# Address translation cache and VM






# Summary memory hierarchy

Hide CPU - memory performance gap
Memory hierarchy with several levels
Principle of locality

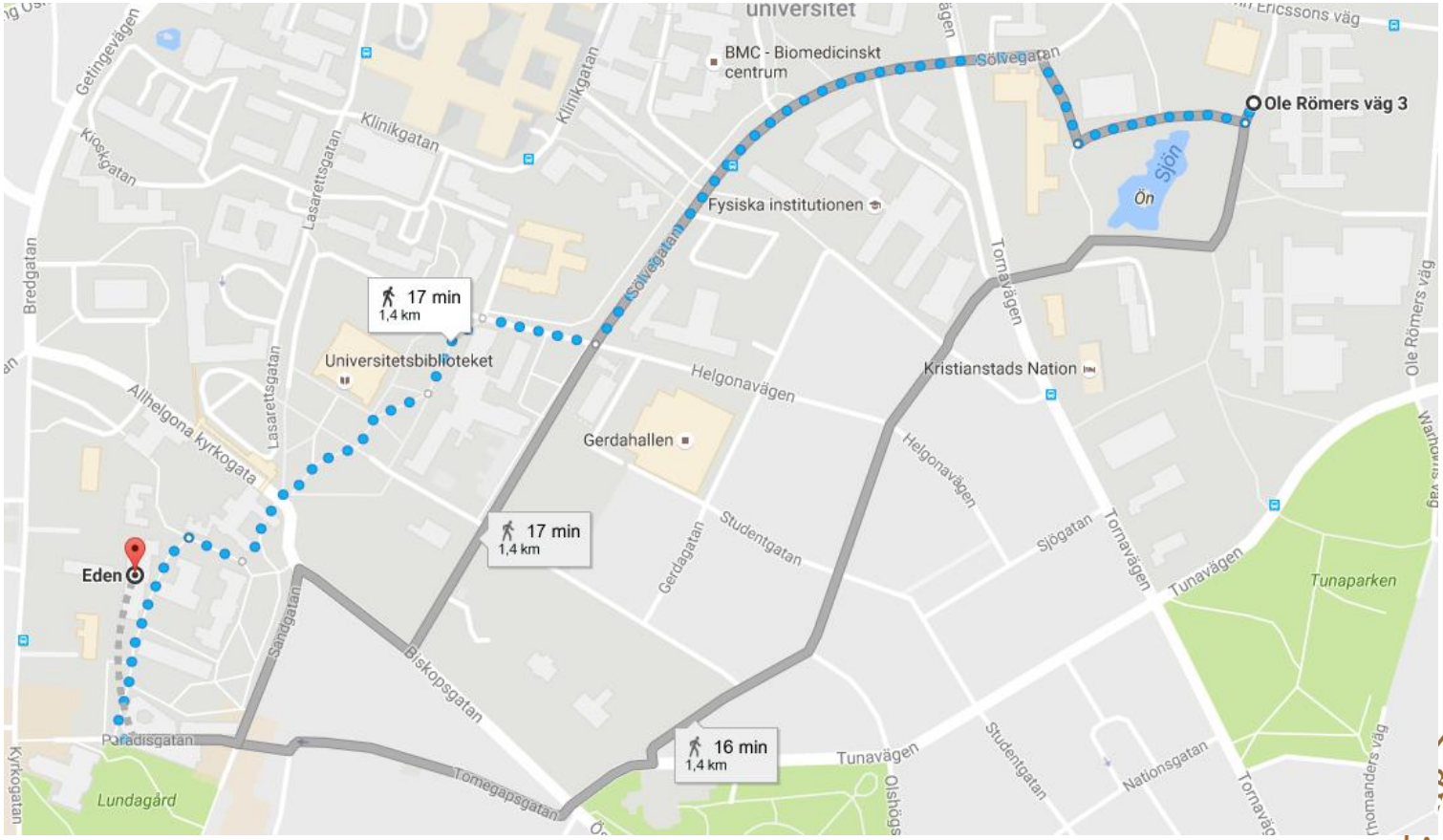
| **Cache memories:** | **Virtual memory:** |
| --- | --- |
| • Fast, small - Close to CPU | • Slow, big - Close to disk |
| • Hardware | • Software |
| • TLB | • TLB |
| • CPU performance equation | • Page-table |
| • Average memory access time | • Very high miss penalty $\Longrightarrow$ miss rate must be low |
| • Optimizations | • Also facilitates: relocation; memory protection; and multiprogramming |

Same 4 design questions - Different answers

# Exam

## Written exam

- 14th Jan., 08-13
- Eden 022, 026, Paradisgatan 5, Hus H
- No mobile phones/Pocket calculator



# Thanks and Good Luck!