Advanced Architecture

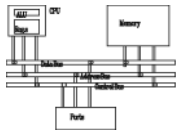
Computer Organization and Assembly Languages

Yung-Yu Chuang

with slides by S. Dandamudi, Peng-Sheng Chen, Kip Irvine, Robert Sedgwick and Kevin Wayne

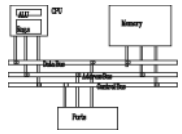
Intel microprocessor history

Early Intel microprocessors



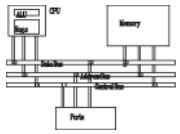
- Intel 8080 (1972)
  - 64K addressable RAM
  - 8-bit registers
  - CP/M operating system
  - 5,6,8,10 MHz
  - 29K transistors
- Intel 8086/8088 (1978) ⟵ my first computer (1986)
  - IBM-PC used 8088
  - 1 MB addressable RAM
  - 16-bit registers
  - 16-bit data bus (8-bit for 8088)
  - separate floating-point unit (8087)
  - used in low-cost microcontrollers now

The IBM-AT



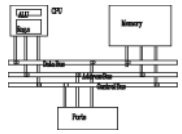
- Intel 80286 (1982)
  - 16 MB addressable RAM
  - Protected memory
  - several times faster than 8086
  - introduced IDE bus architecture
  - 80287 floating point unit
  - Up to 20MHz
  - 134K transistors

Intel IA-32 Family



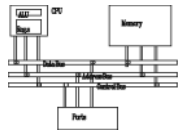
- Intel386 (1985)
  - 4 GB addressable RAM
  - 32-bit registers
  - paging (virtual memory)
  - Up to 33MHz
- Intel486 (1989)
  - instruction pipelining
  - Integrated FPU
  - 8K cache
- Pentium (1993)
  - Superscalar (two parallel pipelines)

Intel P6 Family



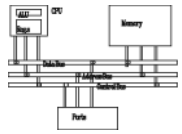
- Pentium Pro (1995)
  - advanced optimization techniques in microcode
  - More pipeline stages
  - On-board L2 cache
- Pentium II (1997)
  - MMX (multimedia) instruction set
  - Up to 450MHz
- Pentium III (1999)
  - SIMD (streaming extensions) instructions (SSE)
  - Up to 1+GHz
- Pentium 4 (2000)
  - NetBurst micro-architecture, tuned for multimedia
  - 3.8+GHz
- Pentium D (2005, Dual core)

IA32 Processors



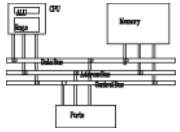
- Totally Dominate Computer Market
- Evolutionary Design
  - Starting in 1978 with 8086
  - Added more features as time goes on
  - Still support old features, although obsolete
- Complex Instruction Set Computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!

ARM history



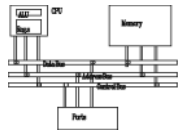
- 1983 developed by Acorn computers
  - To replace 6502 in BBC computers
  - 4-man VLSI design team
  - Its simplicity comes from the inexperience team
  - Match the needs for generalized SoC for reasonable power, performance and die size
  - The first commercial RISC implemenation
- 1990 ARM (Advanced RISC Machine), owned by Acorn, Apple and VLSI

ARM Ltd



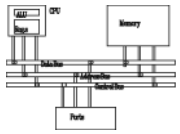
Design and license ARM core design but not fabricate

Why ARM?

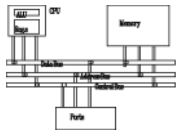


- One of the most licensed and thus widespread processor cores in the world
  - Used in PDA, cell phones, multimedia players, handheld game console, digital TV and cameras
  - ARM7: GBA, iPod
  - ARM9: NDS, PSP, Sony Ericsson, BenQ
  - ARM11: Apple iPhone, Nokia N93, N800
  - 90% of 32-bit embedded RISC processors till 2009
- Used especially in portable devices due to its low power consumption and reasonable performance

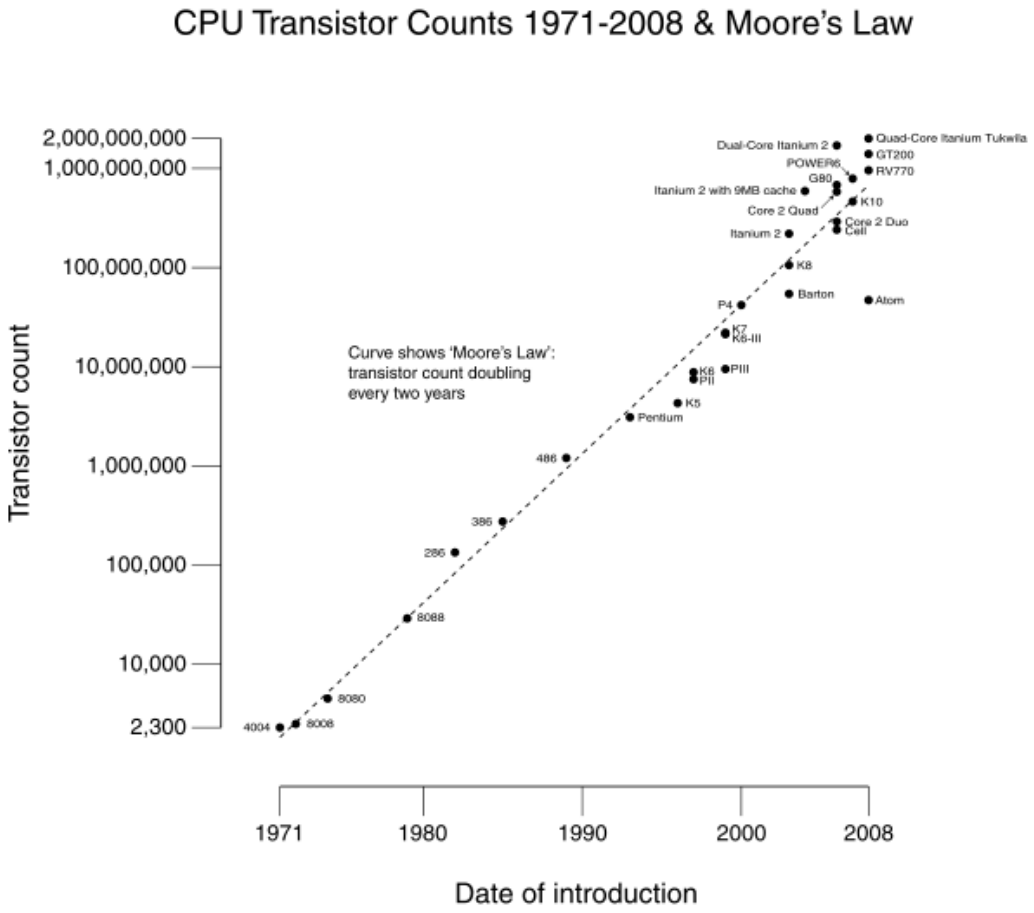
ARM powered products



Performance boost



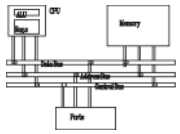
- Increasing clock rate is insufficient. Architecture (pipeline/cache/SIMD) becomes more significant.



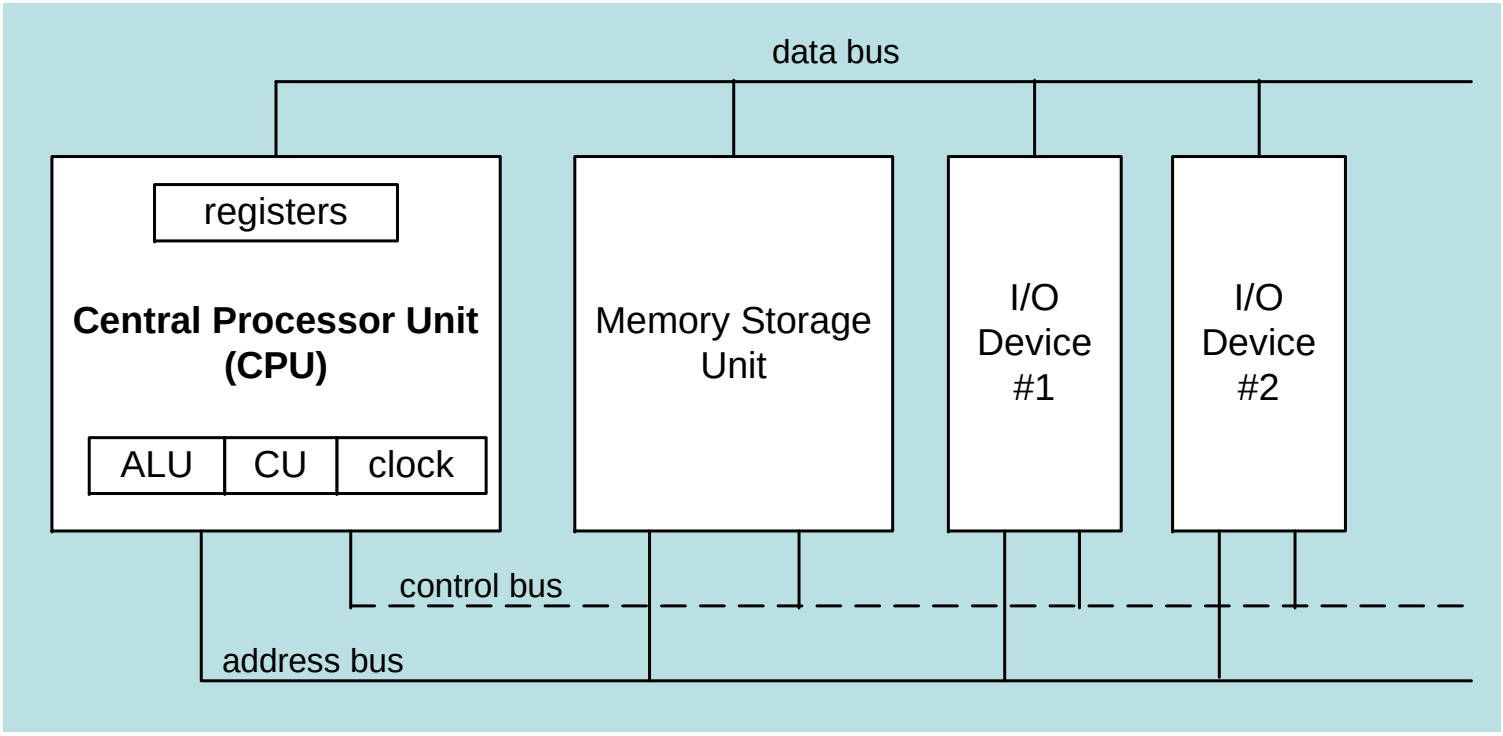
In his 1965 paper, Intel co-founder Gordon Moore observed that "the number of transistors per square inch had doubled every 18 months.

Basic architecture

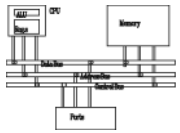
Basic microcomputer design



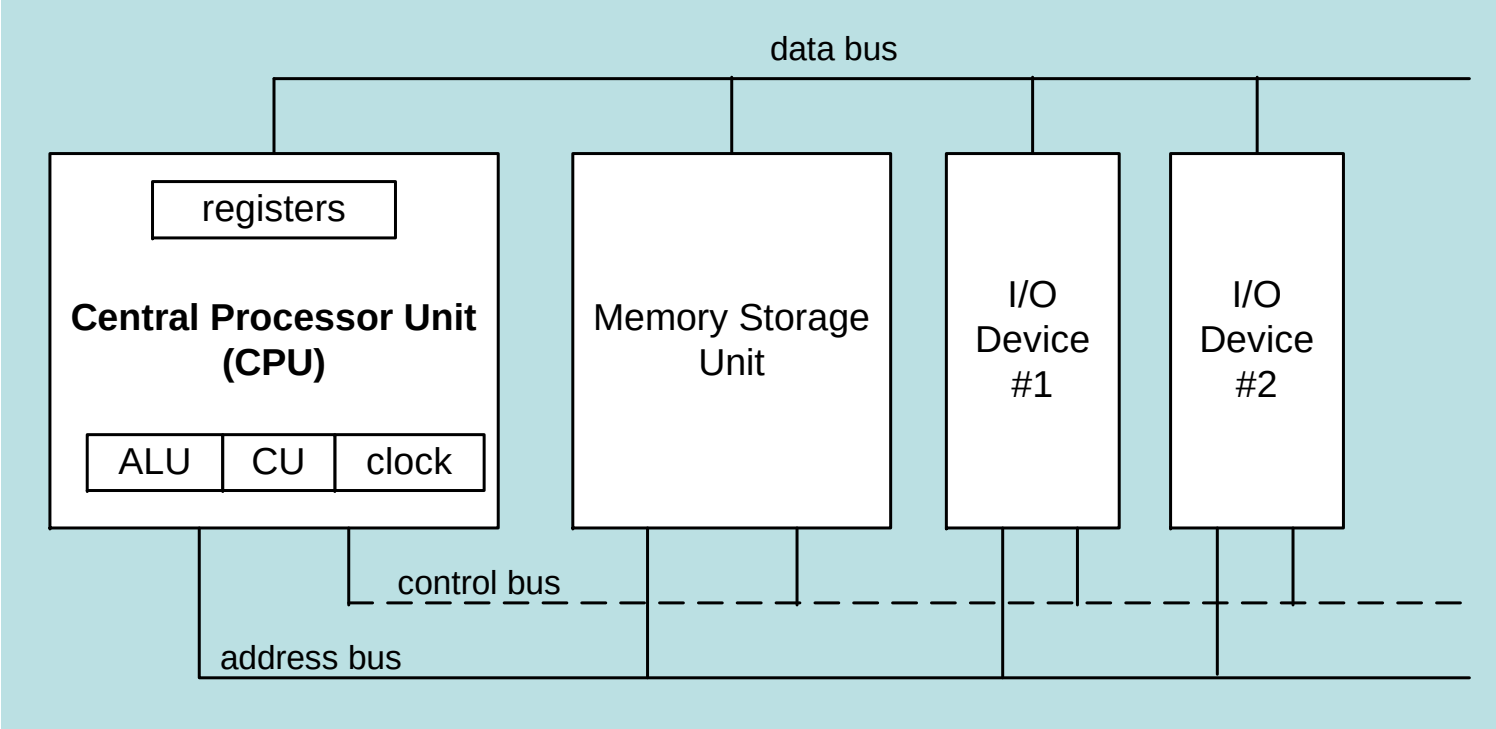
- clock synchronizes CPU operations
- control unit (CU) coordinates sequence of execution steps
- ALU performs arithmetic and logic operations



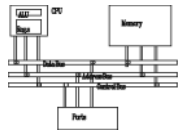
Basic microcomputer design



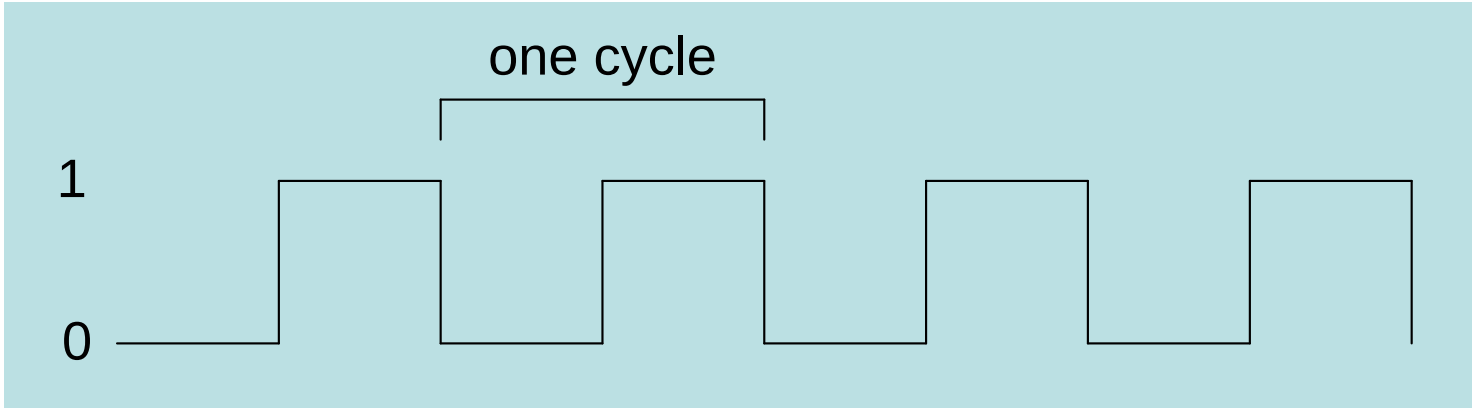
- The memory storage unit holds instructions and data for a running program
- A bus is a group of wires that transfer data from one part to another (data, address, control)



Clock

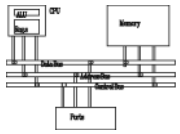


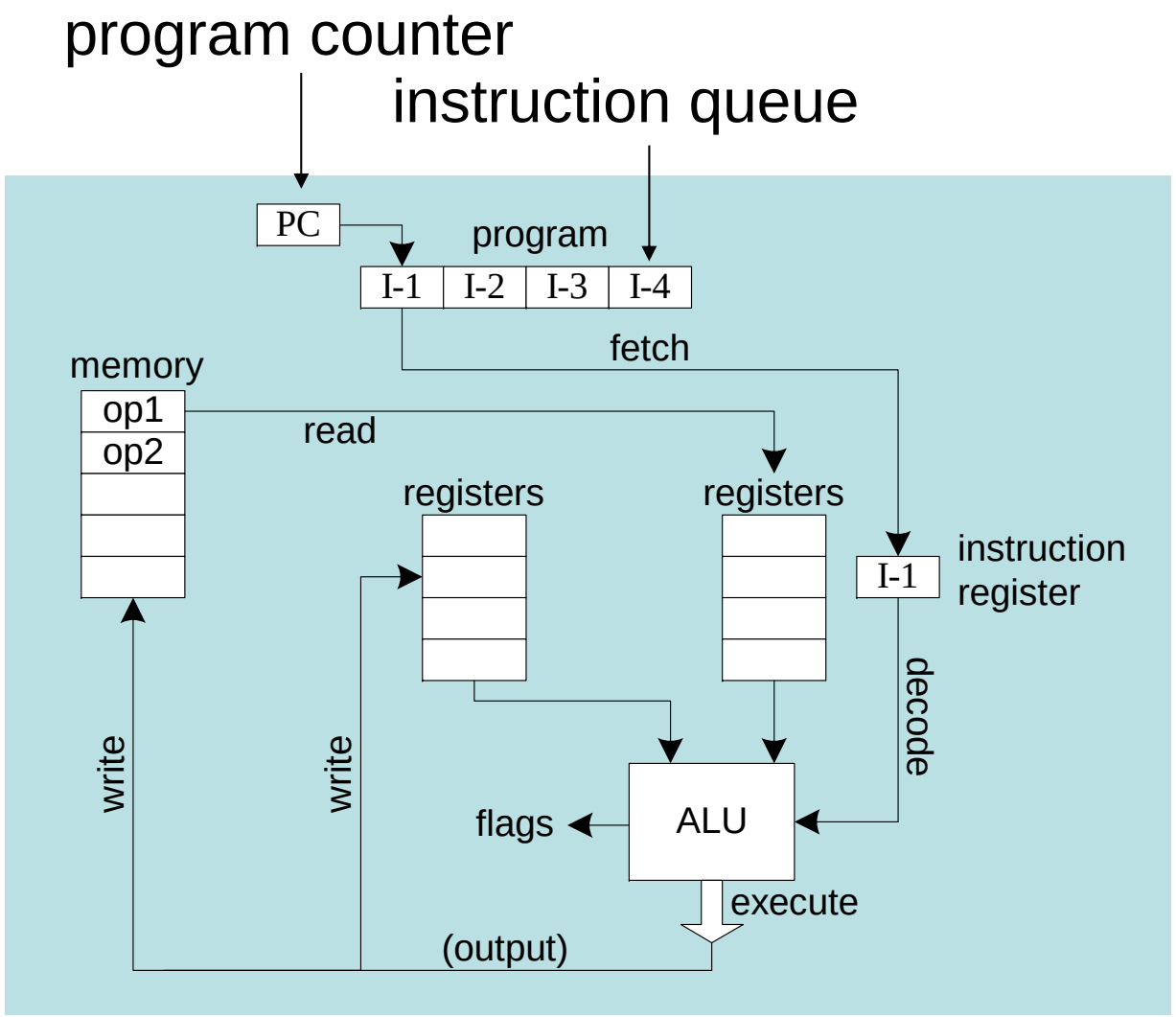
- synchronizes all CPU and BUS operations
- machine (clock) cycle measures time of a single operation
- clock is used to trigger events



- Basic unit of time, 1GHz→clock cycle=1ns
- An instruction could take multiple cycles to complete, e.g. multiply in 8088 takes 50 cycles

Instruction execution cycle

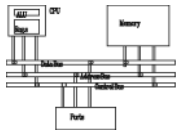




- Fetch
- Decode
- Fetch operands
- Execute
- Store output

Pipeline

Multi-stage pipeline



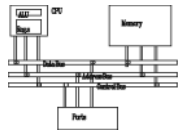
- Pipelining makes it possible for processor to execute instructions in parallel
- Instruction execution divided into discrete stages

Example of a non-pipelined processor. For example, 80386. Many wasted cycles.

Stages

Cycles	S1	S2	S3	S4	S5	S6
1	I-1					
2		I-1				
3			I-1			
4				I-1		
5					I-1	
6						I-1
7	I-2					
8		I-2				
9			I-2			
10				I-2		
11					I-2	
12						I-2

Pipelined execution



- More efficient use of cycles, greater throughput of instructions: (80486 started to use pipelining)

Stages

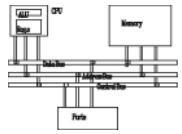
Cycles	S1	S2	S3	S4	S5	S6
1	I-1					
2	I-2	I-1				
3		I-2	I-1			
4			I-2	I-1		
5				I-2	I-1	
6					I-2	I-1
7						I-2

For *k* stages and *n* instructions, the number of required cycles is:

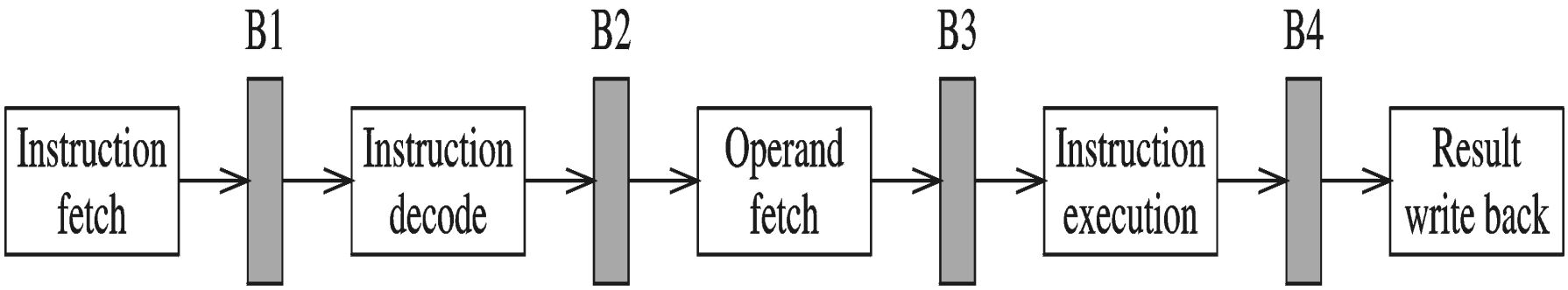
$$k + (n - 1)$$

compared to k*n

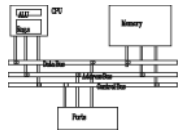
Pipelined execution



- Pipelining requires buffers
  - Each buffer holds a single value
  - Ideal scenario: equal work for each stage
    - Sometimes it is not possible
    - Slowest stage determines the flow rate in the entire pipeline

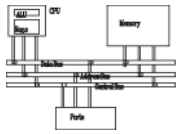


Pipelined execution

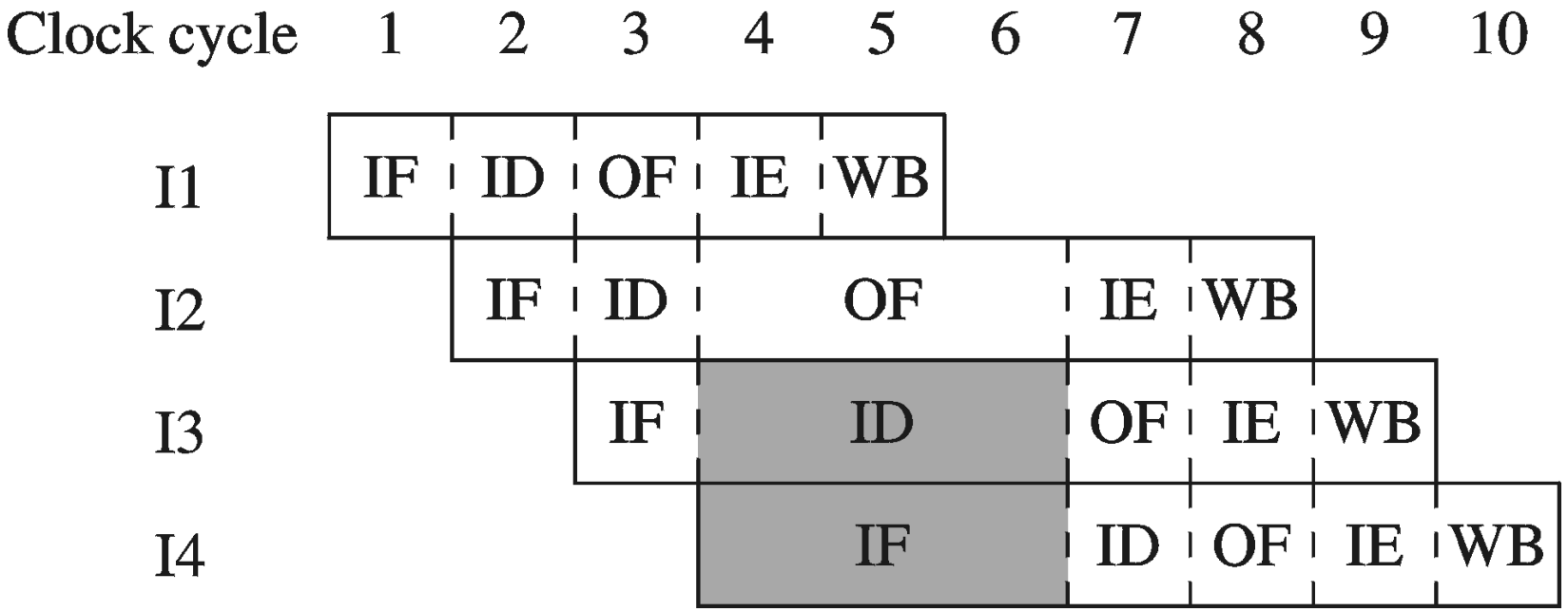


- Some reasons for unequal work stages
  - A complex step cannot be subdivided conveniently
  - An operation takes variable amount of time to execute, e.g. operand fetch time depends on where the operands are located
    - Registers
    - Cache
    - Memory
  - Complexity of operation depends on the type of operation
    - Add: may take one cycle
    - Multiply: may take several cycles

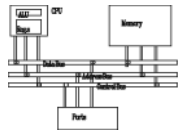
Pipelined execution



- Operand fetch of I2 takes three cycles
  - Pipeline *stalls* for two cycles
    - Caused by hazards
  - Pipeline stalls reduce overall throughput



Wasted cycles (pipelined)



- When one of the stages requires two or more clock cycles, clock cycles are again wasted.

Stages

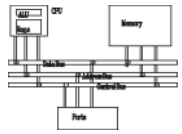
exe

Cycles	S1	S2	S3	S4	S5	S6
1	I-1					
2	I-2	I-1				
3	I-3	I-2	I-1			
4		I-3	I-2	I-1		
5			I-3	I-1		
6				I-2	I-1	
7				I-2		I-1
8				I-3	I-2	
9				I-3		I-2
10					I-3	
11						I-3

For k stages and n instructions, the number of required cycles is:

$$k + (2n - 1)$$

Superscalar



A superscalar processor has multiple execution pipelines. In the following, note that Stage S4 has left and right pipelines (u and v).

Stages

Cycles	S1	S2	S3	S4 (u)	S4 (v)	S5	S6
1	I-1						
2	I-2	I-1					
3	I-3	I-2	I-1				
4	I-4	I-3	I-2	I-1			
5		I-4	I-3	I-1	I-2		
6			I-4	I-3	I-2	I-1	
7				I-3	I-4	I-2	I-1
8					I-4	I-3	I-2
9						I-4	I-3
10							I-4

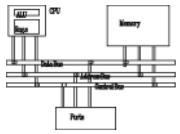
For k states and n instructions, the number of required cycles is:

$$k + n$$

Pentium: 2 pipelines

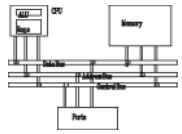
Pentium Pro: 3

Pipeline stages



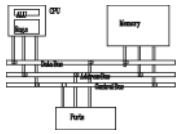
- Pentium 3: 10
- Pentium 4: 20~31
- Next-generation micro-architecture: 14
- ARM7: 3

Hazards

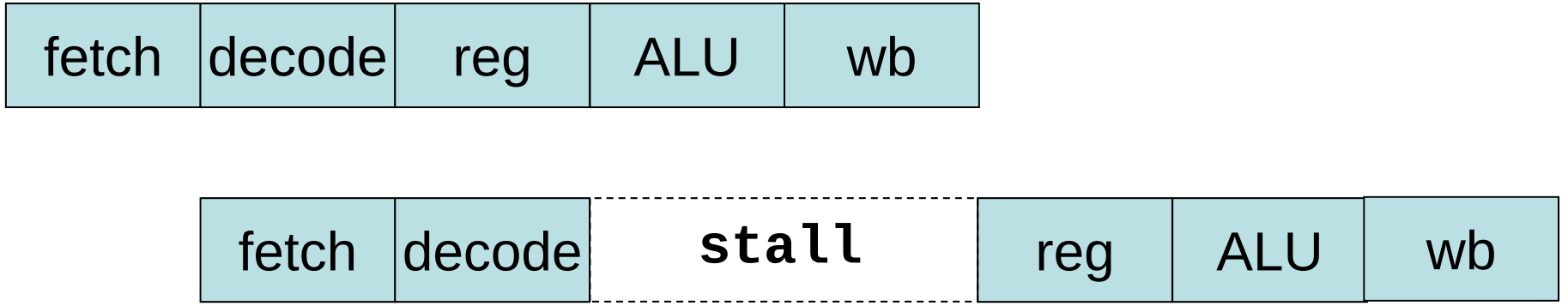


- Three types of hazards
  - Resource hazards
    - Occurs when two or more instructions use the same resource, also called *structural hazards*
  - Data hazards
    - Caused by data dependencies between instructions, e.g. result produced by I1 is read by I2
  - Control hazards
    - Default: sequential execution suits pipelining
    - Altering control flow (e.g., branching) causes problems, introducing control dependencies

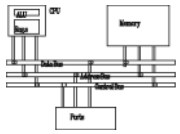
Data hazards



```
add r1, r2, #10      ; write r1
sub r3, r1, #20      ; read r1
```

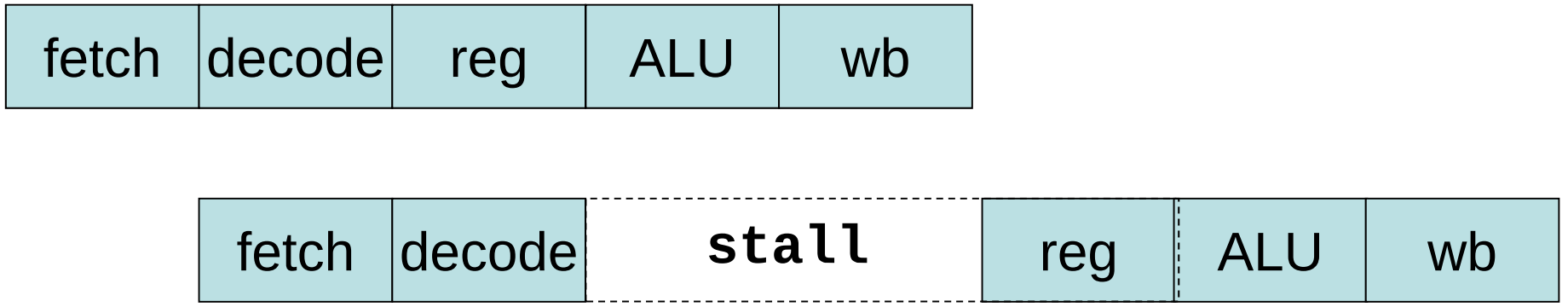


Data hazards

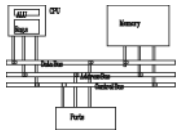


- Forwarding: provides output result as soon as possible

```
add r1, r2, #10      ; write r1
sub r3, r1, #20      ; read r1
```

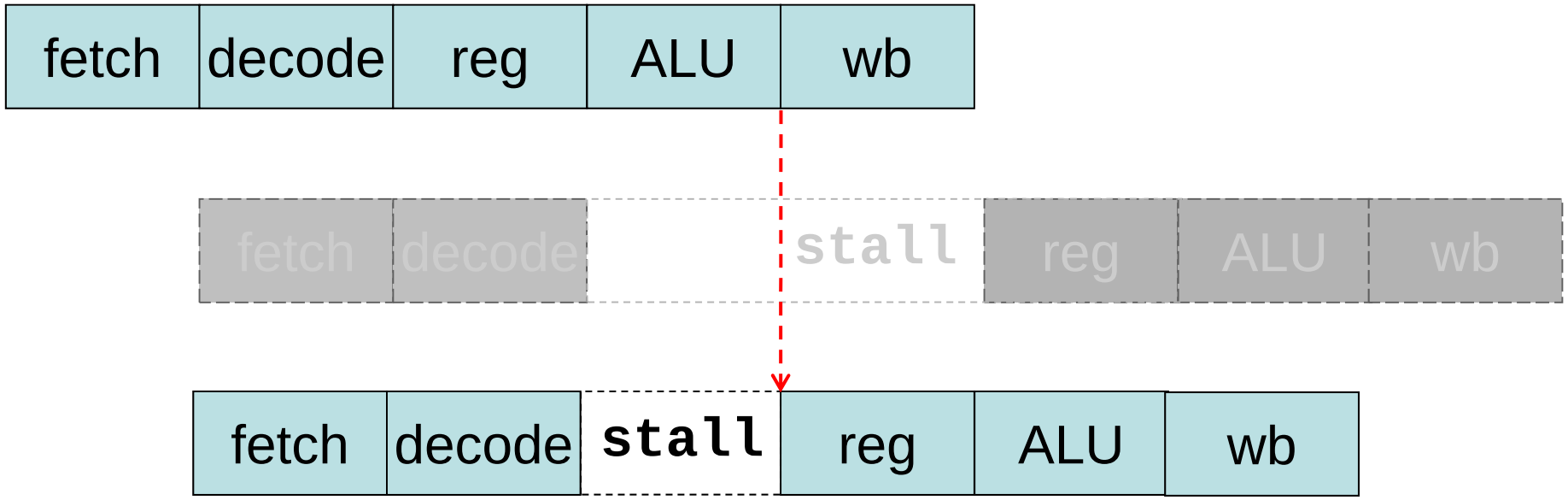


Data hazards

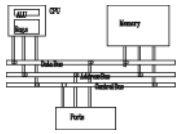


- Forwarding: provides output result as soon as possible

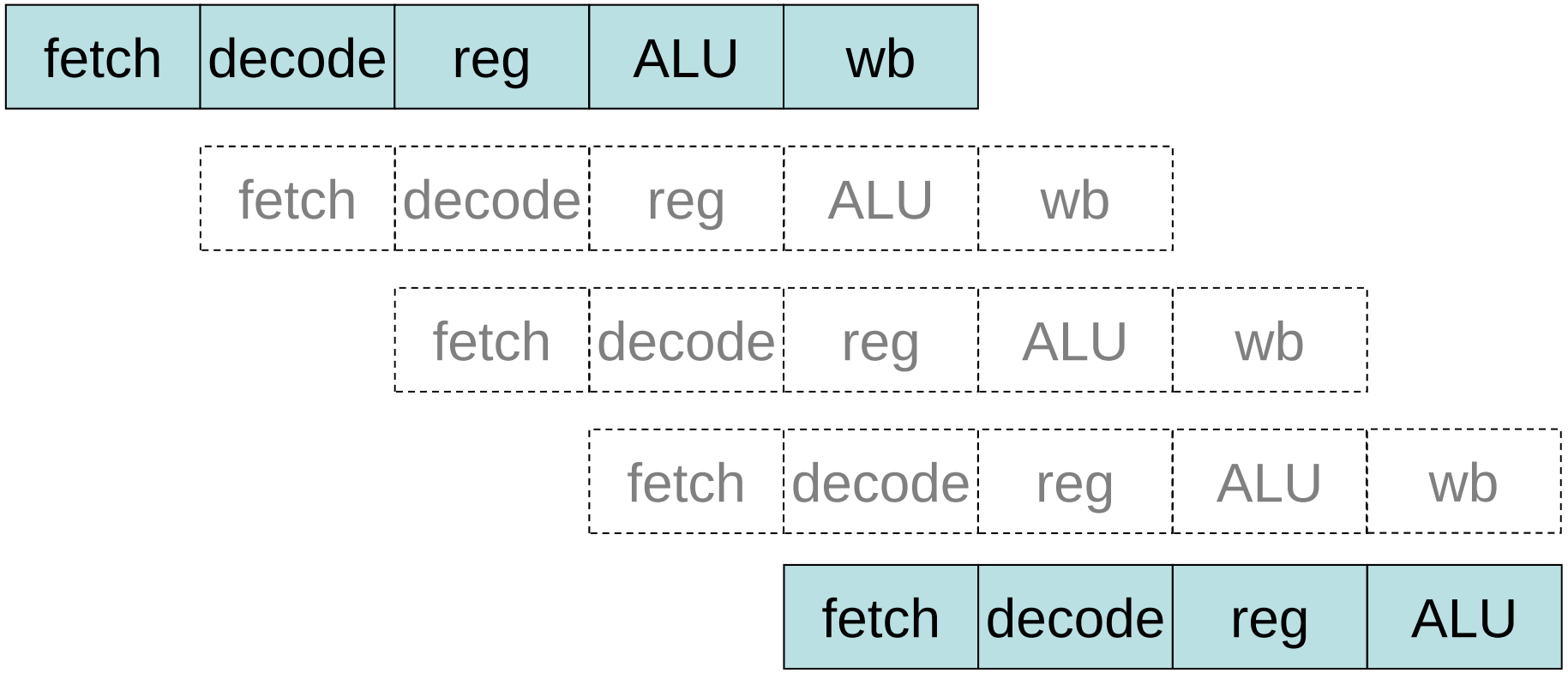
```
add r1, r2, #10      ; write r1
sub r3, r1, #20      ; read r1
```



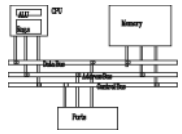
Control hazards



```
            bz   r1, target
            add  r2, r4, 0
            ...
target:     add  r2, r3, 0
```

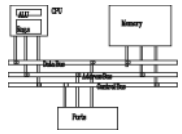


Control hazards



- Braches alter control flow
  - Require special attention in pipelining
  - Need to throw away some instructions in the pipeline
    - Depends on when we know the branch is taken
    - Pipeline wastes three clock cycles
      - Called *branch penalty*
  - Reducing branch penalty
    - Determine branch decision early

Control hazards



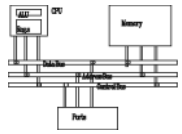
- Delayed branch execution
  - Effectively reduces the branch penalty
  - We always fetch the instruction following the branch
    - Why throw it away?
    - Place a useful instruction to execute
    - This is called *delay slot*

```
add     R2,R3,R4
branch  target
sub     R5,R6,R7
...
```

```
branch  target
add     R2,R3,R4
sub     R5,R6,R7
...
```

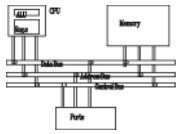
Delay slot

Branch prediction



- Three prediction strategies
  - Fixed
    - Prediction is fixed
      - Example: **branch-never-taken**
        - » Not proper for loop structures
  - Static
    - Strategy depends on the branch type
      - Conditional branch: always not taken
      - Loop: always taken
  - Dynamic
    - Takes run-time history to make more accurate predictions

Branch prediction

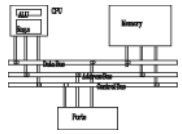


- Static prediction
  - Improves prediction accuracy over Fixed

Instruction type	Instruction Distribution (%)	Prediction: Branch taken?	Correct prediction (%)
Unconditional branch	70*0.4 = 28	Yes	28
Conditional branch	70*0.6 = 42	No	42*0.6 = 25.2
Loop	10	Yes	10*0.9 = 9
Call/return	20	Yes	20

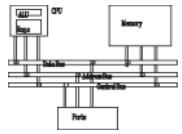
Overall prediction accuracy = **82.2%**

Branch prediction



- Dynamic branch prediction
  - Uses runtime history
    - Takes the past *n* branch executions of the branch type and makes the prediction
  - Simple strategy
    - Prediction of the next branch is the **majority** of the previous *n* branch executions
    - Example*: n* = 3
      - If two or more of the last three branches were taken, the prediction is “branch taken”
    - Depending on the type of mix, we get more than 90% prediction accuracy

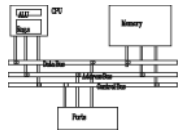
Branch prediction

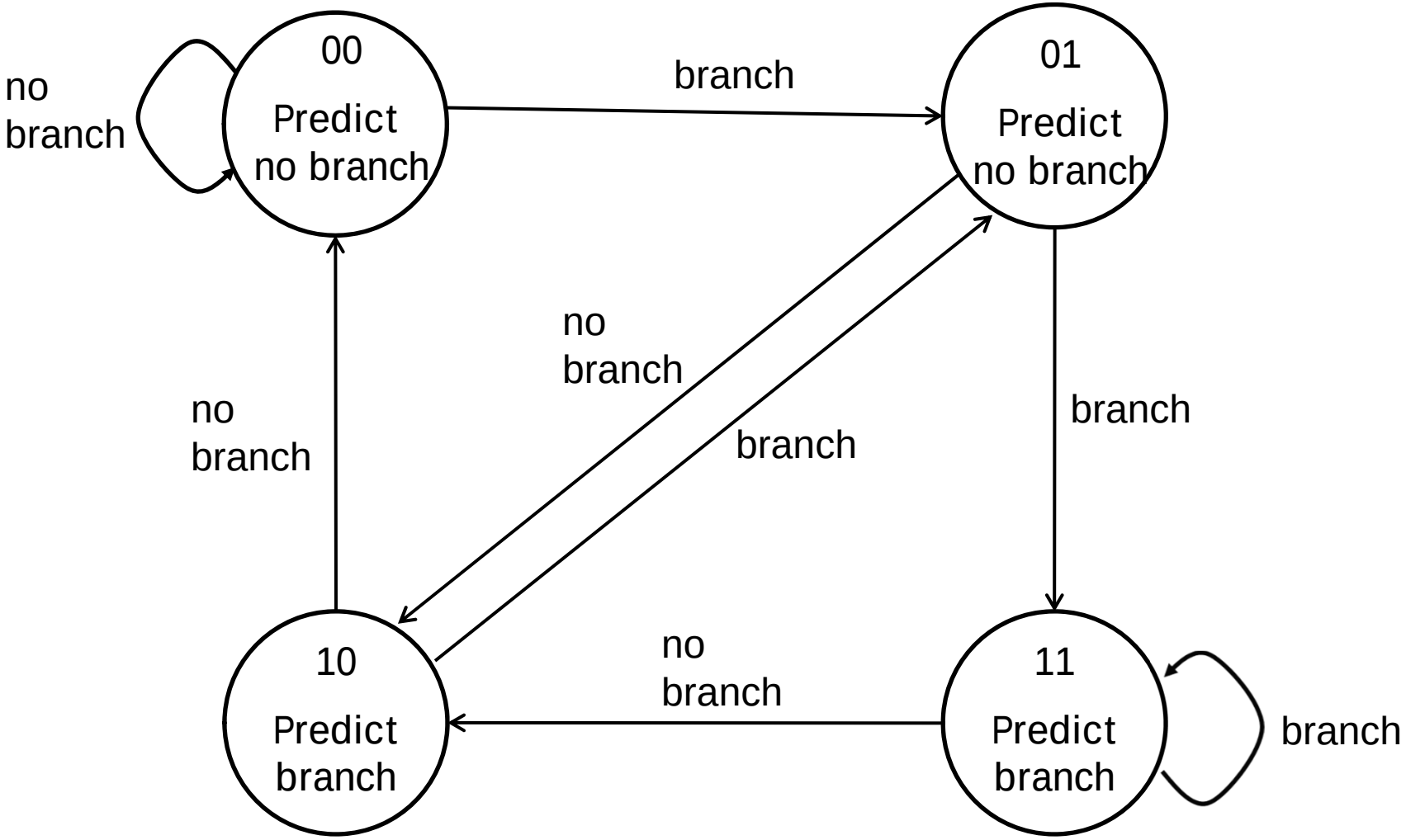


- Impact of past *n* branches on prediction accuracy

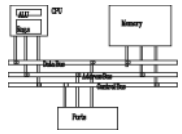
	Type of mix		
n	Compiler	Business	Scientific
0	64.1	64.4	70.4
1	91.9	95.2	86.6
2	93.3	96.5	90.8
3	93.7	96.6	91.0
4	94.5	96.8	91.8
5	94.7	97.0	92.0

Branch prediction





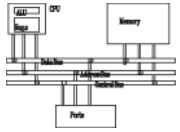
Multitasking

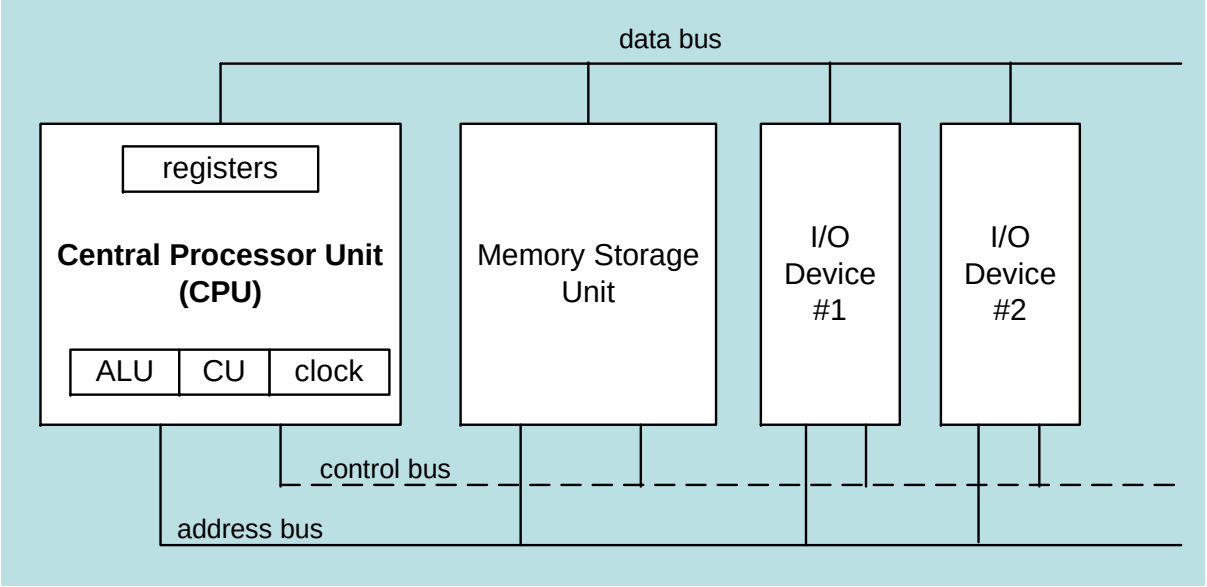


- OS can run multiple programs at the same time.
- Multiple threads of execution within the same program.
- Scheduler utility assigns a given amount of CPU time to each running program.
- Rapid switching of tasks
  - gives illusion that all programs are running at once
  - the processor must support task switching
  - scheduling policy, round-robin, priority

Cache

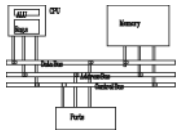
SRAM vs DRAM



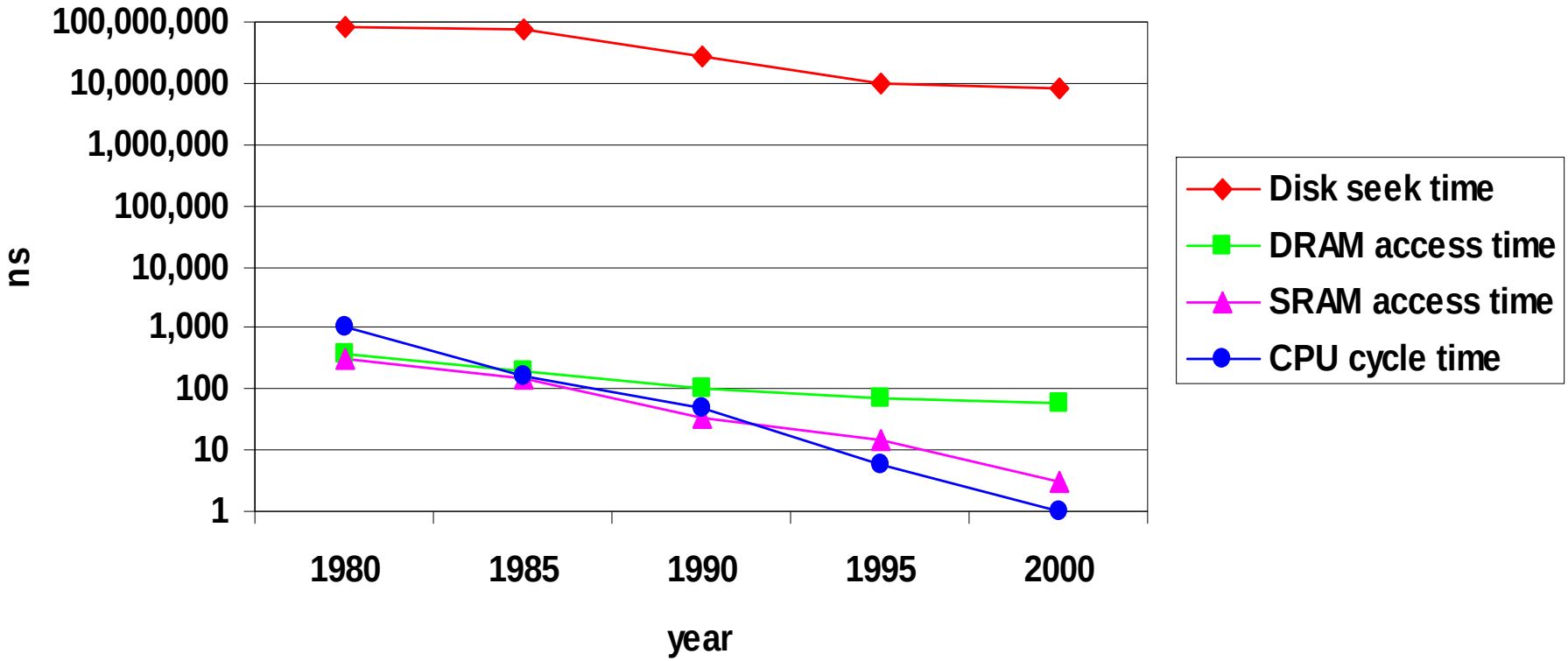


	Tran. per bit	Access time	Needs refresh?	Cost	Applications
SRAM	4 or 6	1X	No	100X	cache memories
DRAM	1	10X	Yes	1X	Main memories, frame buffers

The CPU-Memory gap

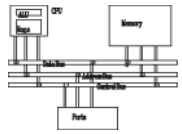


The gap widens between DRAM, disk, and CPU speeds.



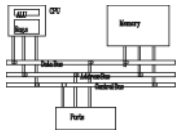
	register	cache	memory	disk
Access time (cycles)	1	1-10	50-100	20,000,000

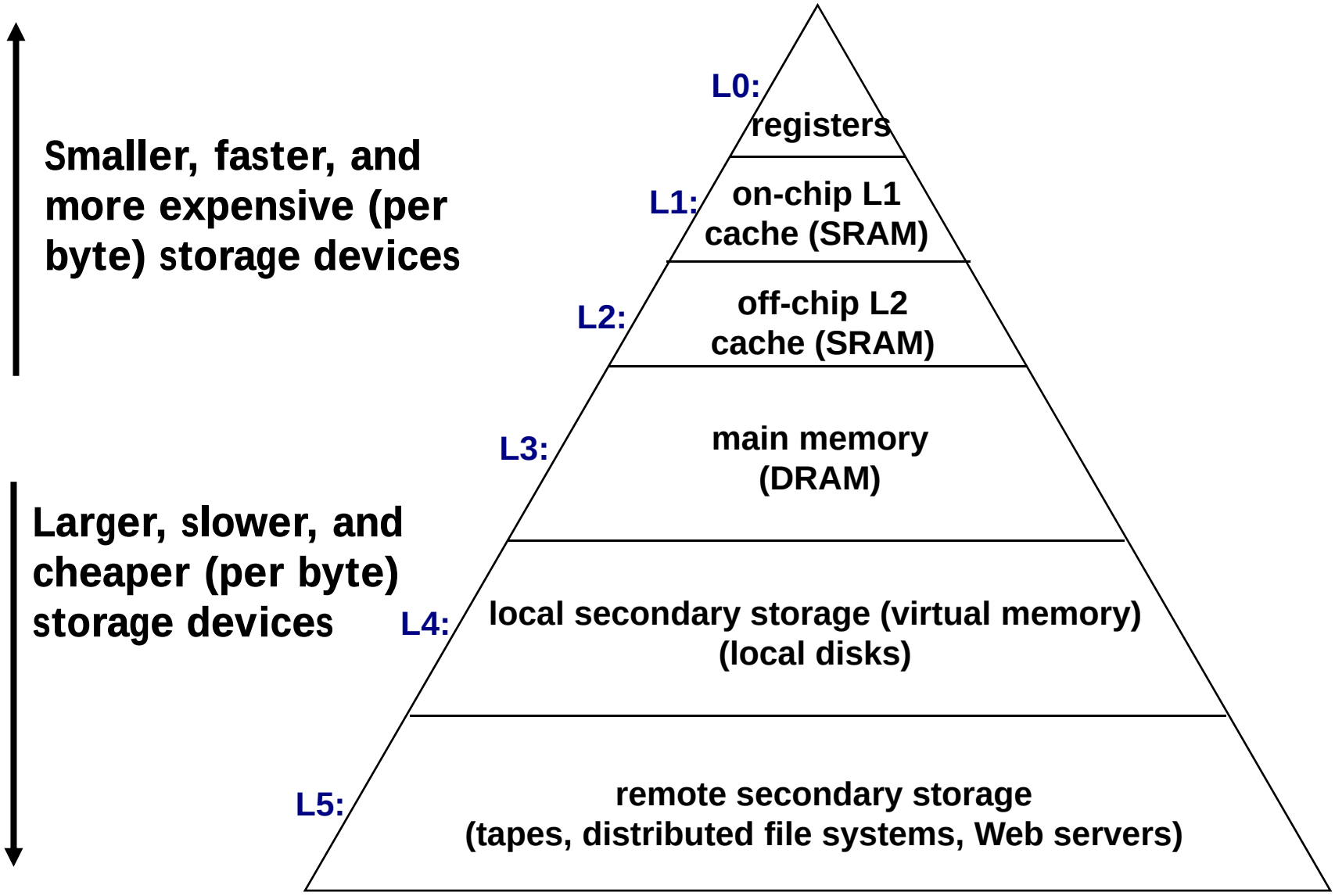
Memory hierarchies



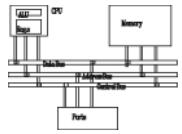
- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.
- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.

Memory system in practice

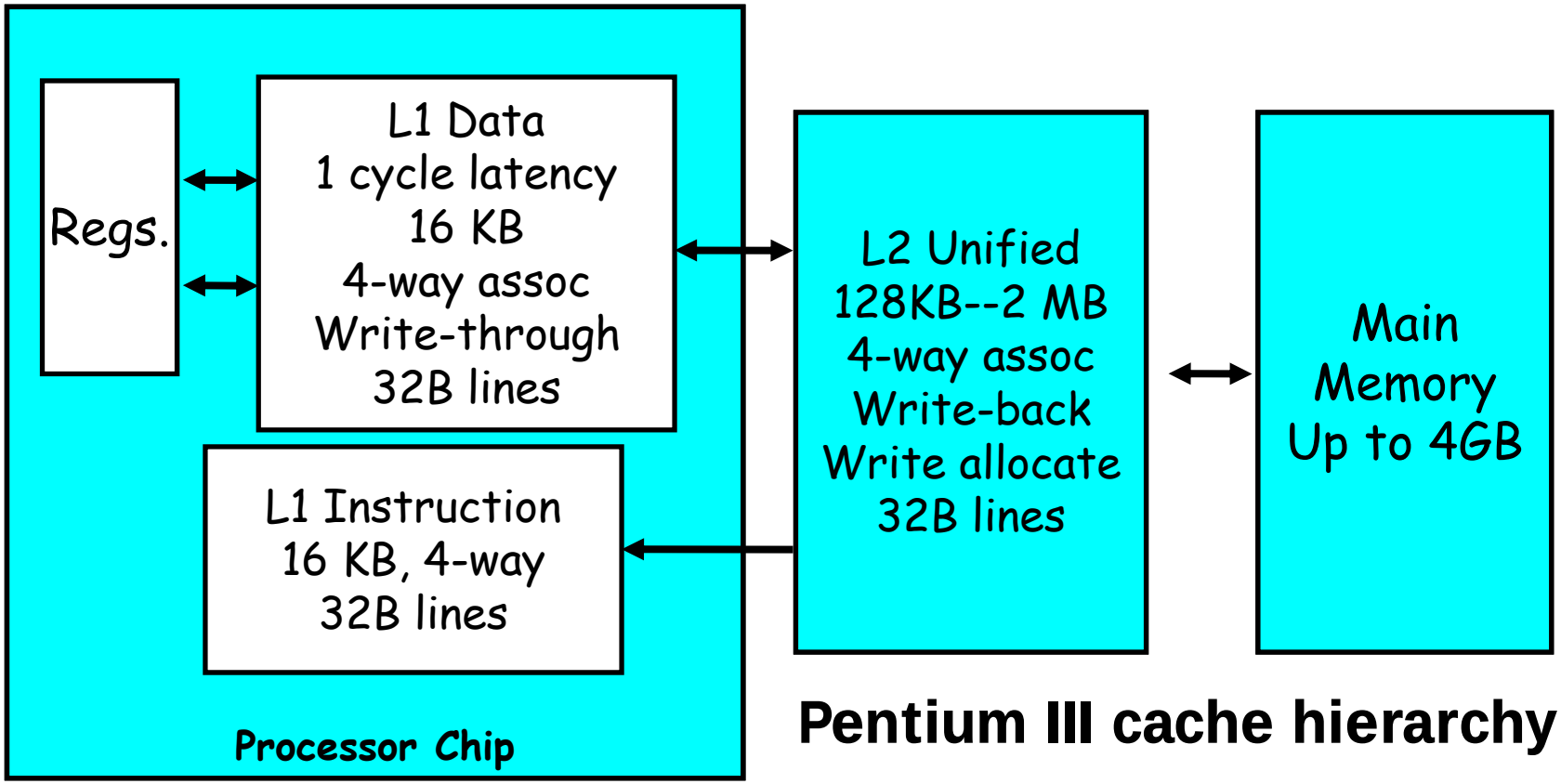




Reading from memory

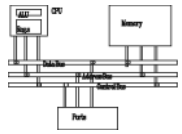


- Multiple machine cycles are required when reading from memory, because it responds much more slowly than the CPU (e.g.33 MHz). The wasted clock cycles are called wait states.



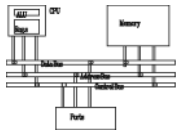
Pentium III cache hierarchy

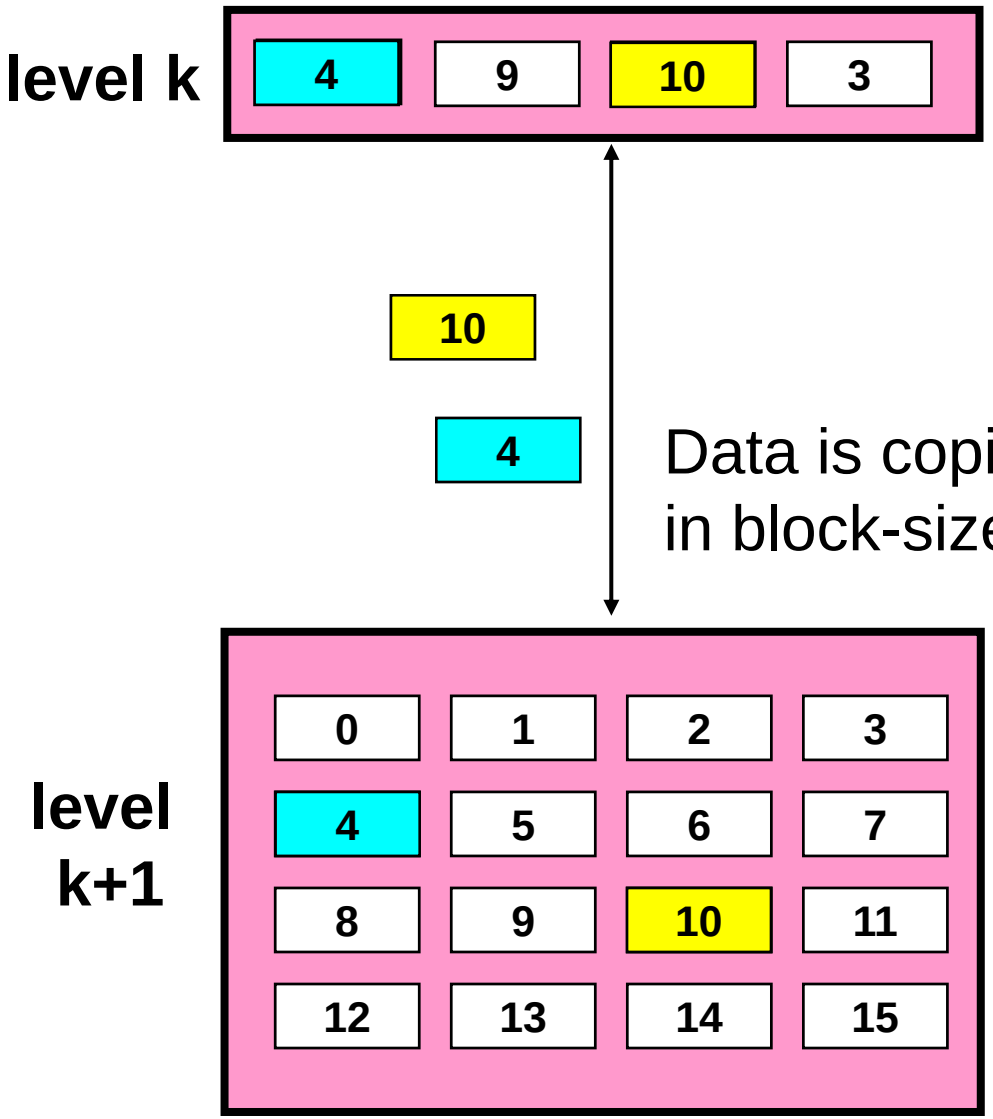
Cache memory



- High-speed expensive static RAM both inside and outside the CPU.
  - Level-1 cache: inside the CPU
  - Level-2 cache: outside the CPU
- Cache hit: when data to be read is already in cache memory
- Cache miss: when data to be read is not in cache memory. When? compulsory, capacity and conflict.
- Cache design: cache size, n-way, block size, replacement policy

Caching in a memory hierarchy

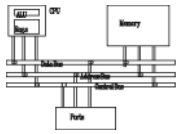


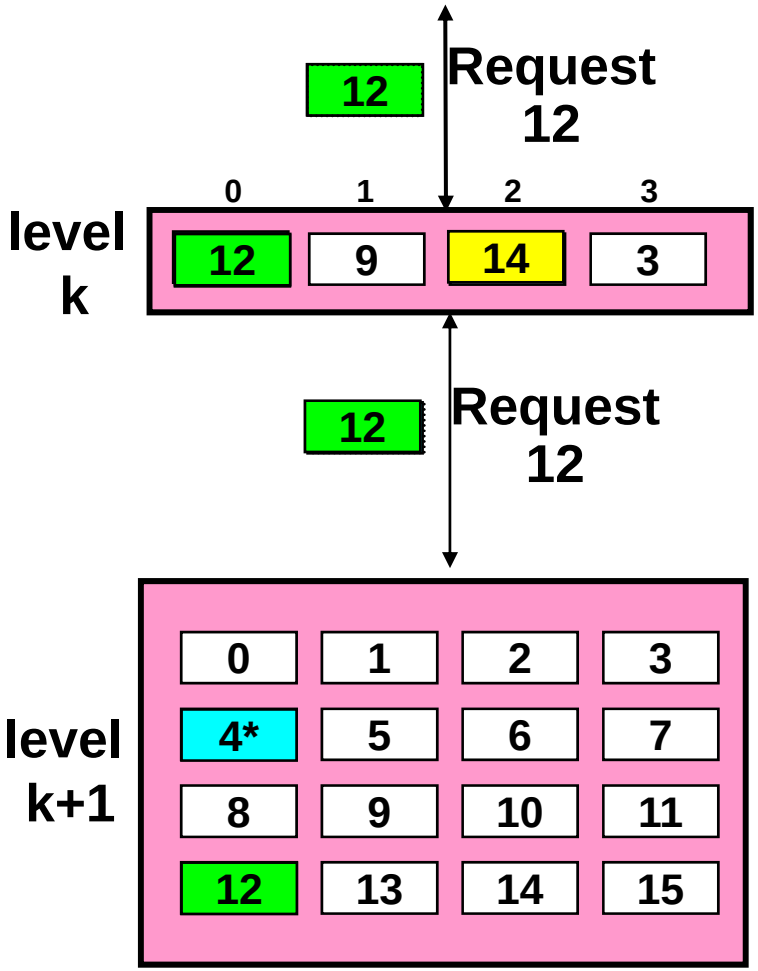


Smaller, faster, more
Expensive device at
level k caches a
subset of the blocks
from level k+1

Larger, slower, cheaper
Storage device at level
k+1 is partitioned into
blocks.

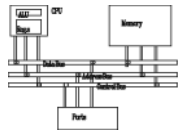
General caching concepts





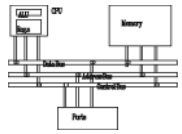
- Program needs object d, which is stored in some block b.
- Cache hit
  - Program finds b in the cache at level k. E.g., block 14.
- Cache miss
  - b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
  - If level k cache is full, then some current block must be replaced (evicted). Which one is the "victim"?
    - Placement policy: where can the new block go? E.g., b mod 4
    - Replacement policy: which block should be evicted? E.g., LRU

Locality



- Principle of Locality: programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - Temporal locality: recently referenced items are likely to be referenced in the near future.
  - Spatial locality: items with nearby addresses tend to be referenced close together in time.
- In general, *programs with good locality run faster then programs with poor locality*
- Locality is the reason why cache and virtual memory are designed in architecture and operating system. Another example is web browser caches recently visited webpages.

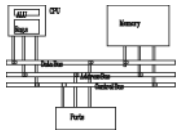
Locality example



```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
return sum;
```

- **Data**
  - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
  - Reference sum each iteration: **Temporal locality**
- **Instructions**
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**

Locality example

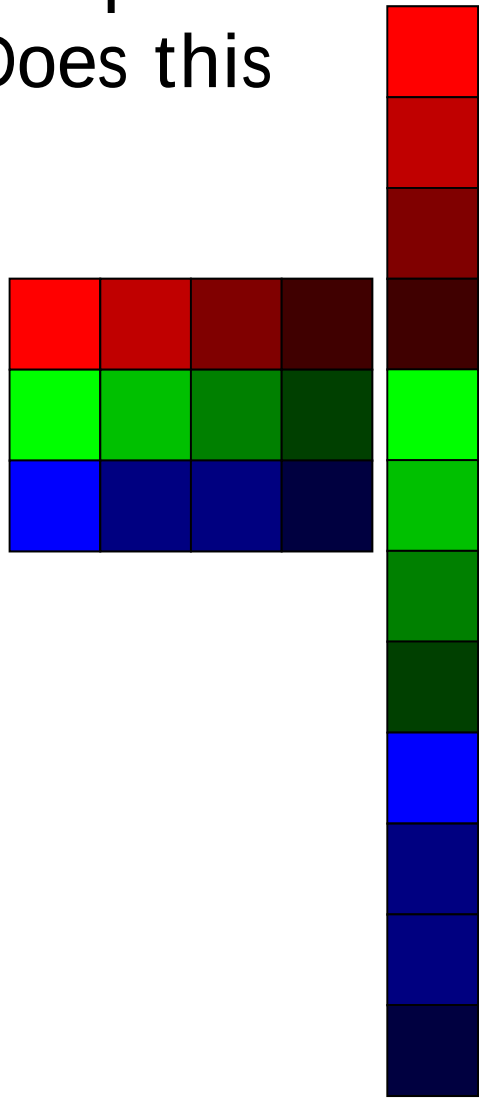


- Being able to look at code and get a qualitative sense of its locality is important. Does this function have good locality?

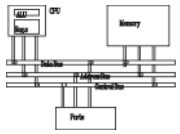
```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

stride-1 reference pattern



Locality example

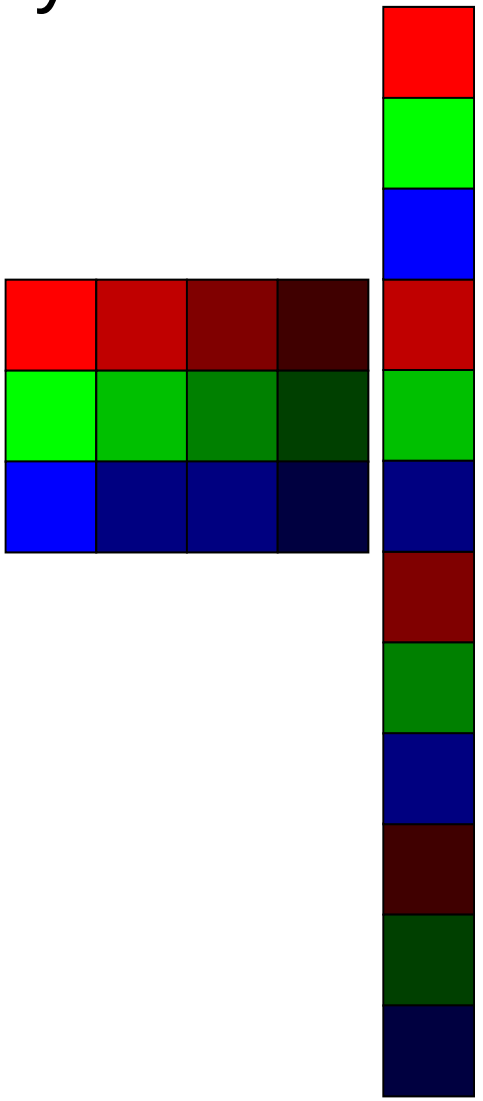


- Does this function have good locality?

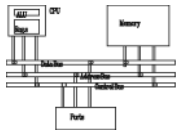
```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

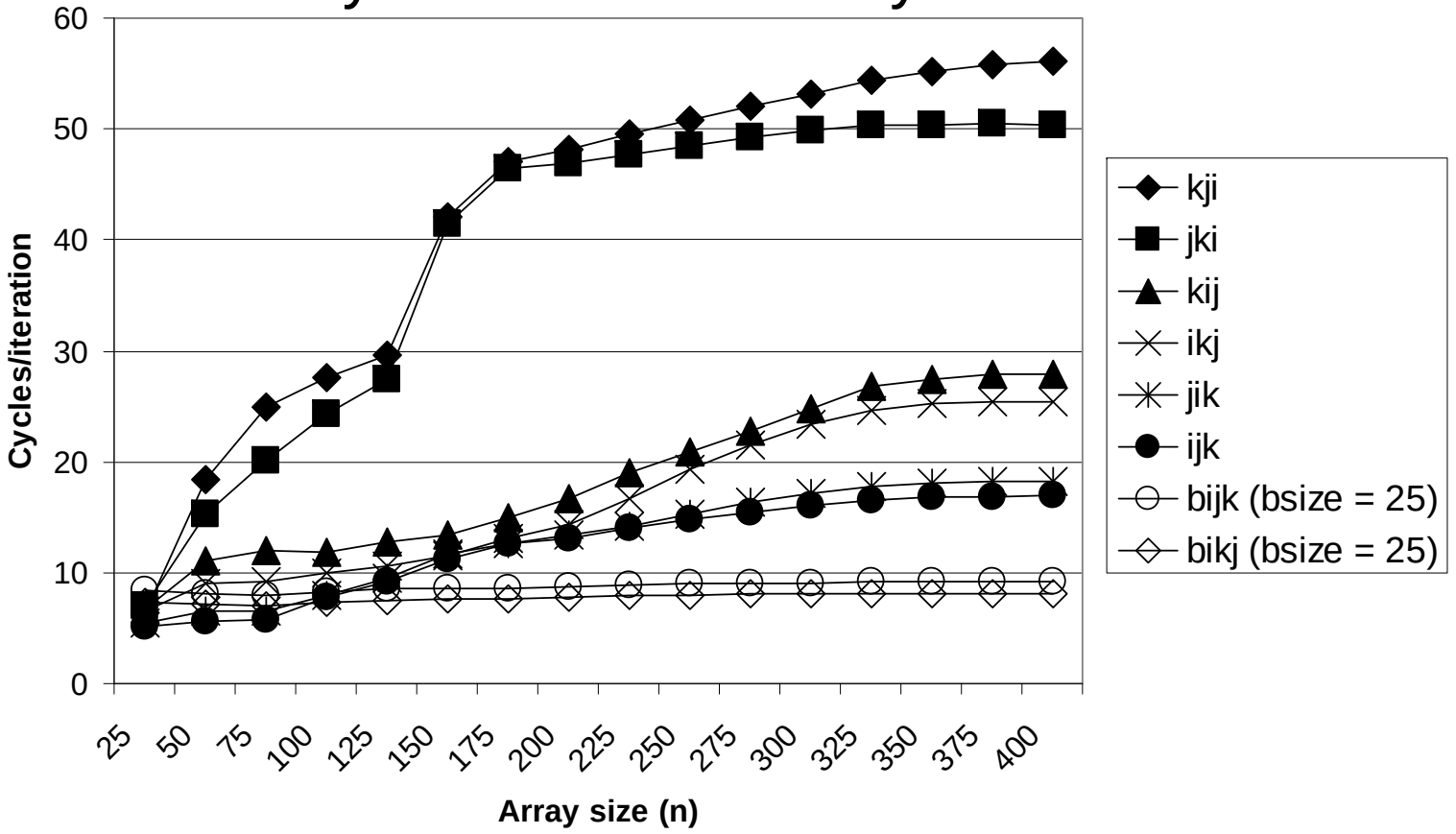
stride-N reference pattern



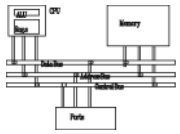
Blocked matrix multiply performance



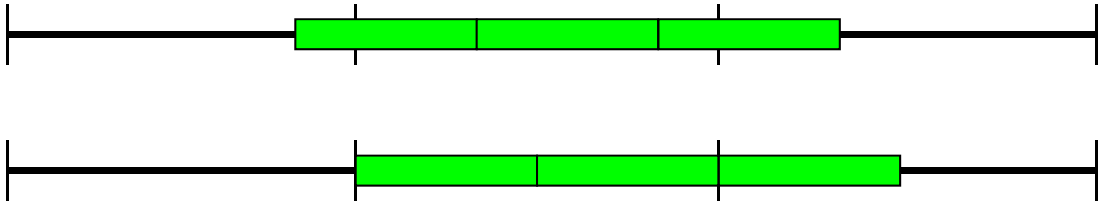
- Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)
  - relatively insensitive to array size.



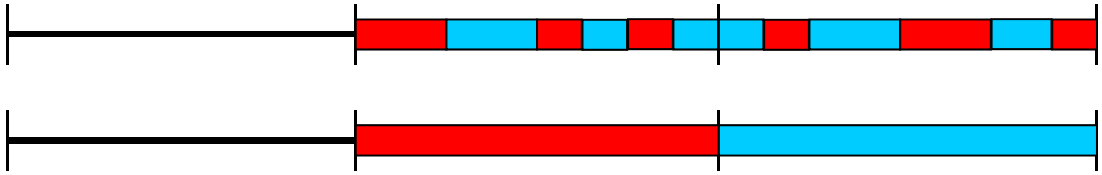
Cache-conscious programming



- make sure that memory is cache-aligned



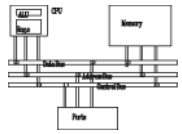
- Split data into hot and cold (list example)



- Use union and bitfields to reduce size and increase locality

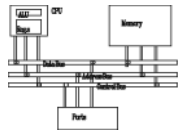
SIMD

SIMD

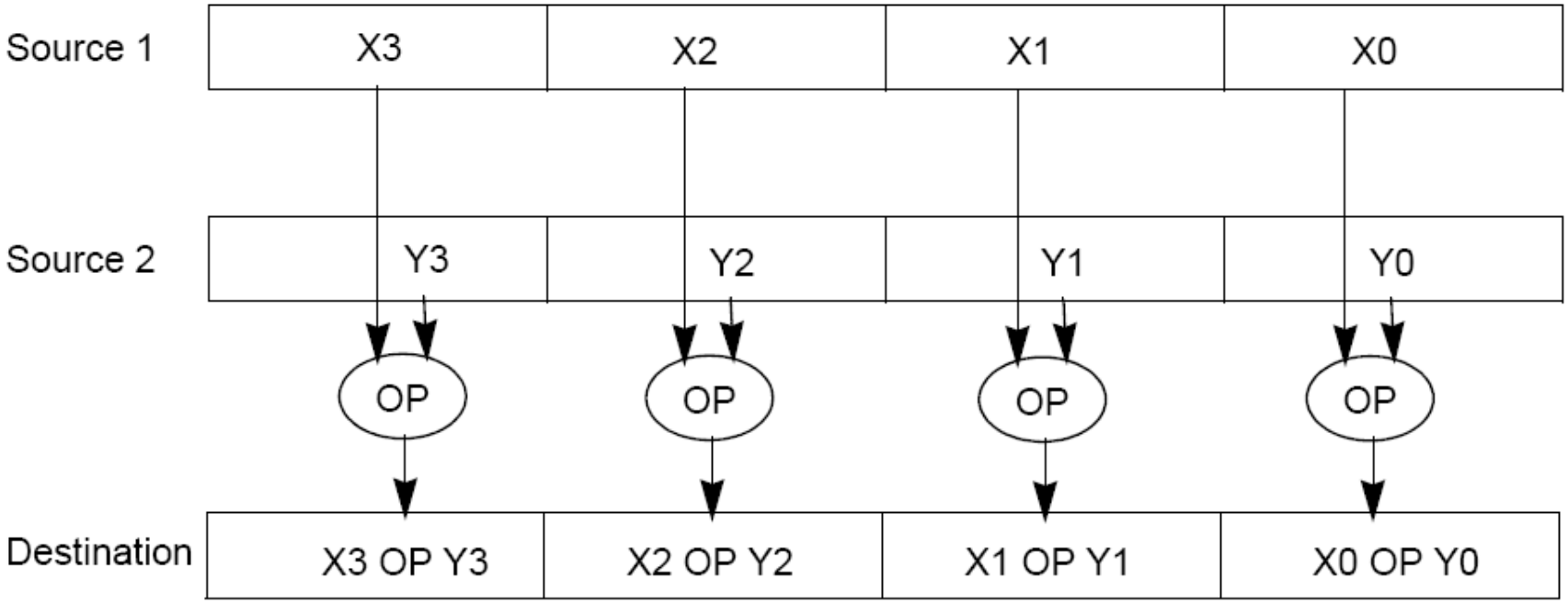


- MMX (Multimedia Extension) was introduced in 1996 (Pentium with MMX and Pentium II).
- Intel analyzed multimedia applications and found they share the following characteristics:
  - Small native data types (8-bit pixel, 16-bit audio)
  - Recurring operations
  - Inherent parallelism

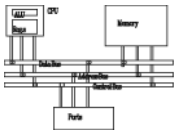
SIMD

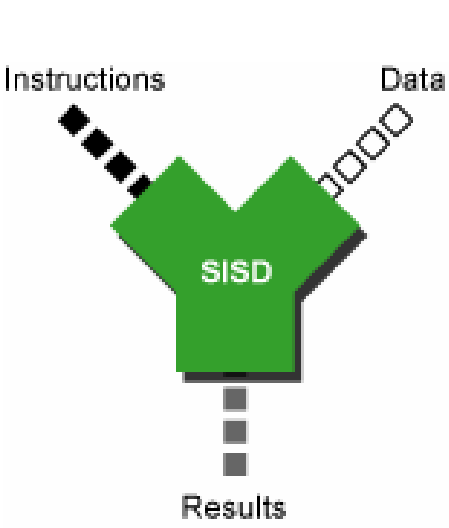


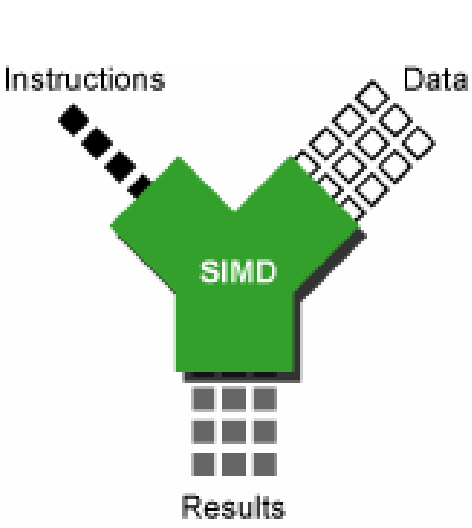
- SIMD (single instruction multiple data) architecture performs the same operation on multiple data elements in parallel
- **`PADDW MM0, MM1`**

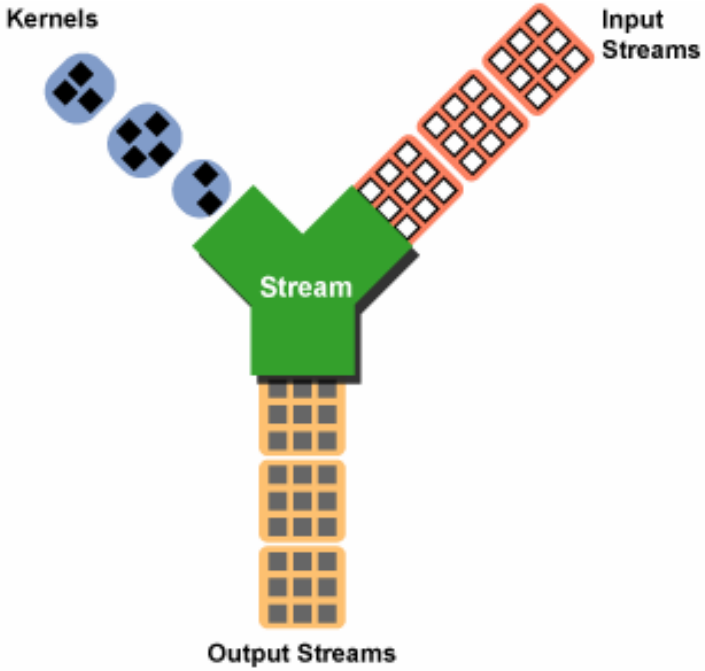


SISD/SIMD/Streaming

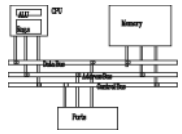






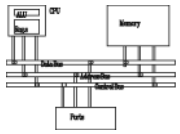


MMX

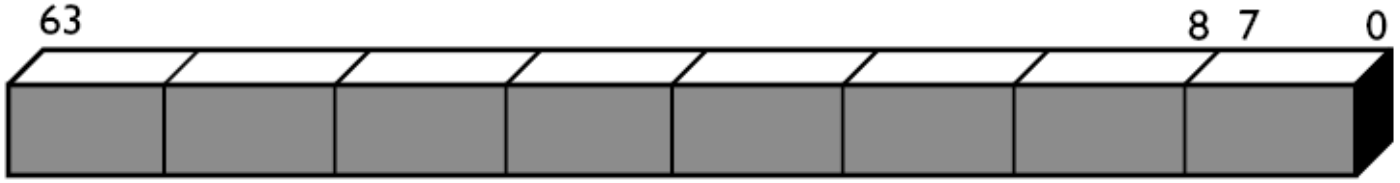


- After analyzing a lot of existing applications such as graphics, MPEG, music, speech recognition, game, image processing, they found that many multimedia algorithms execute the same instructions on many pieces of data in a large data set.
- Typical elements are small, 8 bits for pixels, 16 bits for audio, 32 bits for graphics and general computing.
- New data type: 64-bit packed data type. Why 64 bits?
  - Good enough
  - Practical

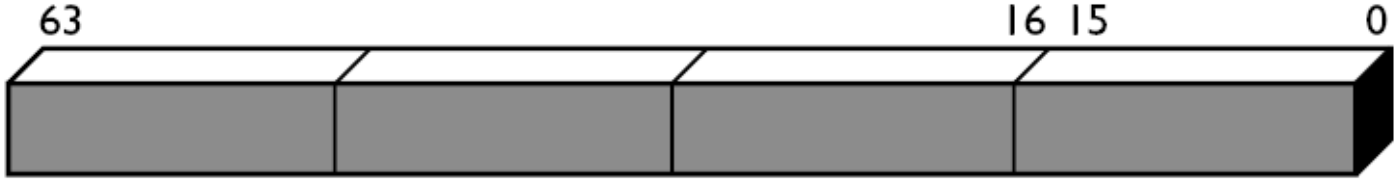
MMX data types



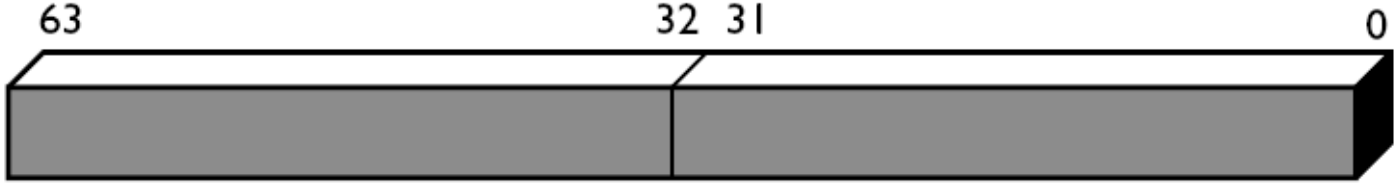
Packed Byte: 8 bytes packed into 64 bits



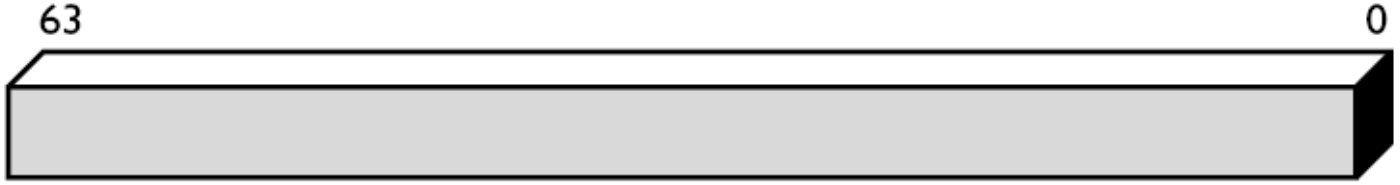
Packed Word: 4 words packed into 64 bits



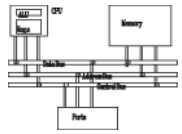
Packed Doubleword: 2 doublewords packed into 64 bits



Packed Quadword: One 64-bit quantity

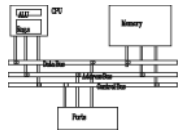


MMX instructions

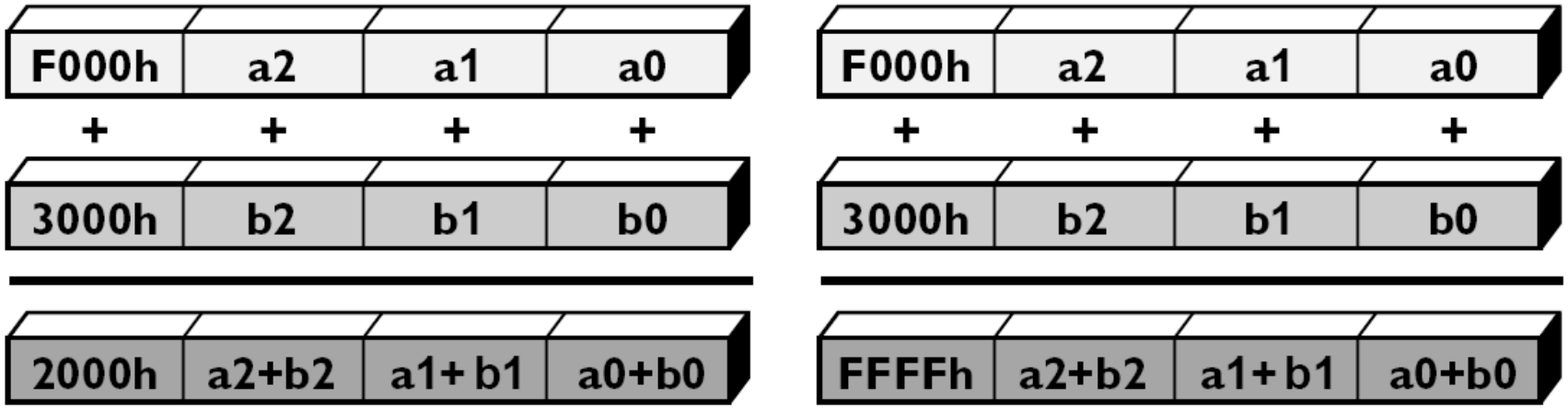


- 57 MMX instructions are defined to perform the parallel operations on multiple data elements packed into 64-bit data types.
- These include **`add`**, **`subtract`**, **`multiply`**, **`compare`**, and **`shift`**, **`data conversion`**, **`64-bit data move`**, **`64-bit logical operation`** and **`multiply-add`** for multiply-accumulate operations.
- All instructions except for data move use MMX registers as operands.
- Most complete support for 16-bit operations.

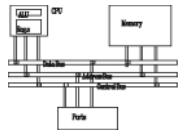
Saturation arithmetic



- Useful in graphics applications.
- When an operation overflows or underflows, the result becomes the largest or smallest possible representable number.
- Two types: signed and unsigned saturation

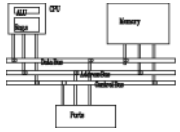


Keys to SIMD programming

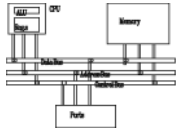


- Efficient data layout
- Elimination of branches

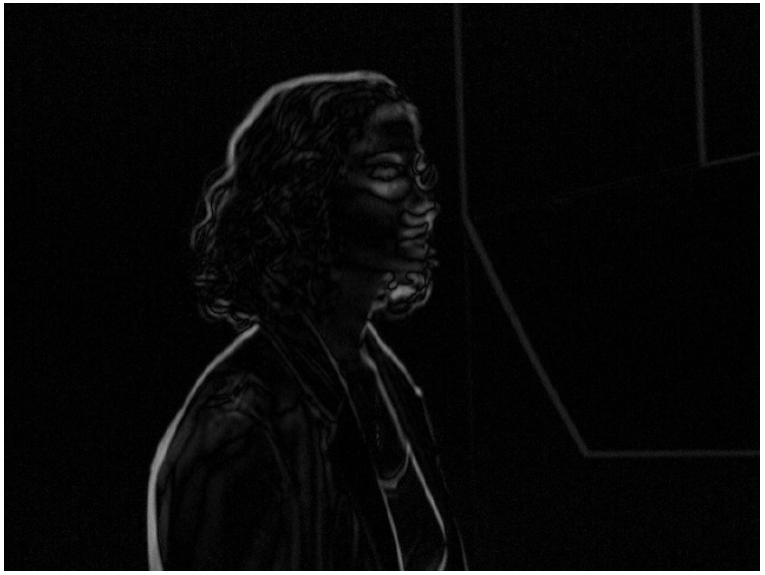
Application: frame difference



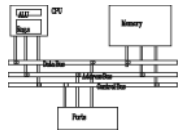
Application: frame difference



(A-B) or (B-A)

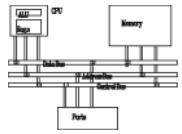


Application: frame difference



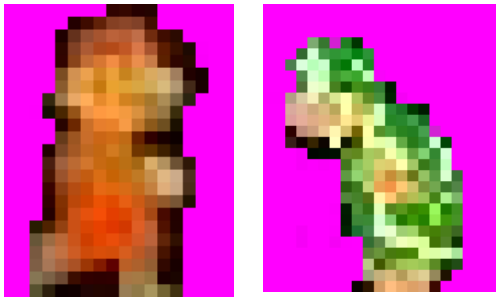
```
MOVQ      mm1, A //move 8 pixels of image A
MOVQ      mm2, B //move 8 pixels of image B
MOVQ      mm3, mm1 // mm3=A
PSUBSB    mm1, mm2 // mm1=A-B
PSUBSB    mm2, mm3 // mm2=B-A
POR       mm1, mm2 // mm1=|A-B|
```

Data-independent computation



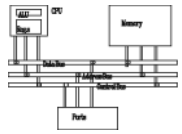
- Each operation can execute without needing to know the results of a previous operation.
- Example, sprite overlay

```
for i=1 to sprite_Size
   if  sprite[i]=clr
   then out_color[i]=bg[i]
   else out_color[i]=sprite[i]
```



- How to execute data-dependent calculations on several pixels in parallel.

Application: sprite overlay



Phase 1

a3	a2	a1	a0
=	=	=	=
clear_color	clear_color	clear_color	clear_color

1111…1111	0000…0000	1111…1111	0000…0000

Phase 2

a3	a2	a1	a0

c3	c2	c1	c0

A and (Complement of Mask)

0000…0000	1111…1111	0000…0000	1111…1111

C and Mask

1111…1111	0000…0000	1111…1111	0000…0000

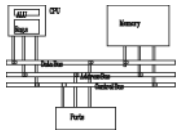
0	a2	0	a0

c3	0	c1	0

OR the two results to finish the overlay

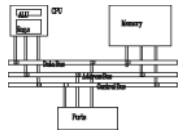
c3	a2	c1	a0

Application: sprite overlay

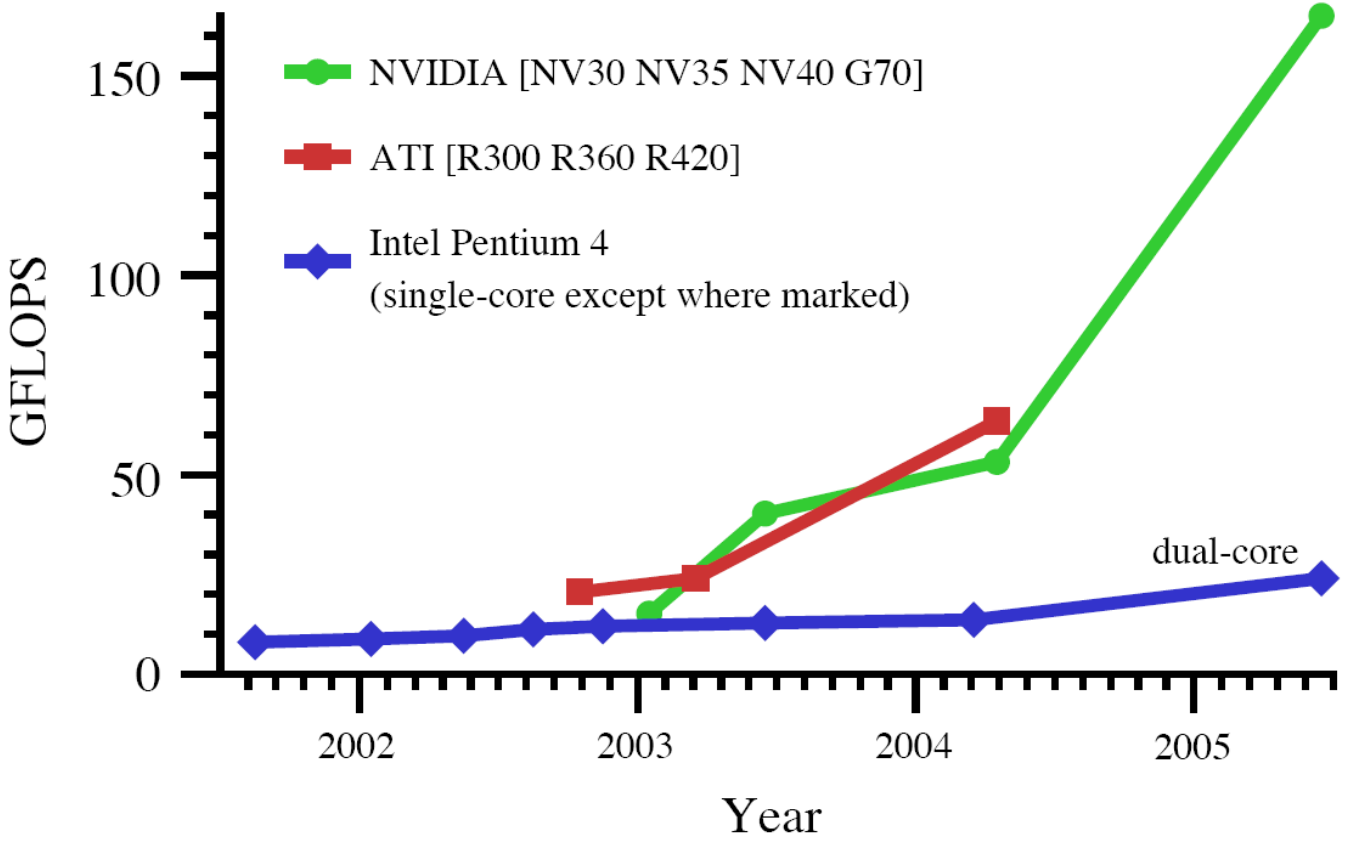


```
MOVQ     mm0, sprite
MOVQ     mm2, mm0
MOVQ     mm4, bg
MOVQ     mm1, clr
PCMPEQW  mm0, mm1
PAND     mm4, mm0
PANDN    mm0, mm2
POR      mm0, mm4
```

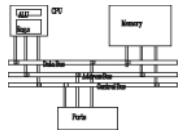
Other SIMD architectures



- Graphics Processing Unit (GPU): nVidia 7800, 24 pipelines (8 vector/16 fragment)



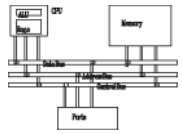
Impacts on programming



- You need to be aware of architecture issues to write more efficient programs (such as cache-aware).
- Need parallel thinking for better utilizing parallel features of processors.

RISC v.s. CISC

Trade-offs of instruction sets



high-level language —**compiler**→ **machine code**

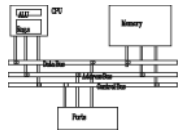
C, C++

Lisp, Prolog, Haskell…

semantic gap

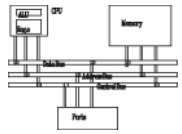
- Before 1980, the trend is to increase instruction complexity (one-to-one mapping if possible) to bridge the gap. Reduce fetch from memory. Selling point: number of instructions, addressing modes. (CISC)
- 1980, RISC. Simplify and regularize instructions to introduce advanced architecture for better performance, pipeline, cache, superscalar.

RISC



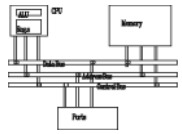
- 1980, Patternson and Ditzel (Berkeley),RISC
- Features
  - Fixed-length instructions
  - Load-store architecture
  - Register file
- Organization
  - Hard-wired logic
  - Single-cycle instruction
  - Pipeline
- Pros: small die size, short development time, high performance
- Cons: low code density, not x86 compatible

RISC Design Principles



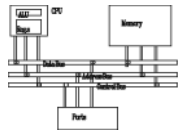
- Simple operations
  - Simple instructions that can execute in one cycle
- Register-to-register operations
  - Only load and store operations access memory
  - Rest of the operations on a register-to-register basis
- Simple addressing modes
  - A few addressing modes (1 or 2)
- Large number of registers
  - Needed to support register-to-register operations
  - Minimize the procedure call and return overhead

RISC Design Principles



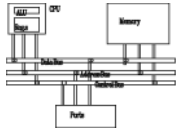
- Fixed-length instructions
  - Facilitates efficient instruction execution
- Simple instruction format
  - Fixed boundaries for various fields
    - opcode, source operands,...

CISC and RISC



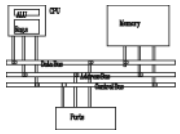
- CISC – complex instruction set
  - large instruction set
  - high-level operations (simpler for compiler?)
  - requires microcode interpreter (could take a long time)
  - examples: Intel 80x86 family
- RISC – reduced instruction set
  - small instruction set
  - simple, atomic instructions
  - directly executed by hardware very quickly
  - easier to incorporate advanced architecture design
  - examples: ARM (Advanced RISC Machines) and DEC Alpha (now Compaq), PowerPC, MIPS

CISC and RISC



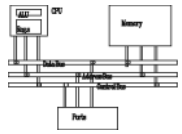
	CISC (Intel 486)	RISC (MIPS R4000)
#instructions	235	94
Addr. modes	11	1
Inst. Size (bytes)	1-12	4
GP registers	8	32

Why RISC?



- Simple instructions are preferred
  - Complex instructions are mostly ignored by compilers
    - Due to semantic gap
- Simple data structures
  - Complex data structures are used relatively infrequently
  - Better to support a few simple data types efficiently
    - Synthesize complex ones
- Simple addressing modes
  - Complex addressing modes lead to variable length instructions
    - Lead to inefficient instruction decoding and scheduling

Why RISC? (cont'd)



- Large register set
  - Efficient support for procedure calls and returns
    - Patterson and Sequin's study
      - Procedure call/return: 12–15% of HLL statements
        - » Constitute 31–33% of machine language instructions
        - » Generate nearly half (45%) of memory references
  - Small activation record
    - Tanenbaum's study
      - Only 1.25% of the calls have more than 6 arguments
      - More than 93% have less than 6 local scalar variables
      - Large register set can avoid memory references