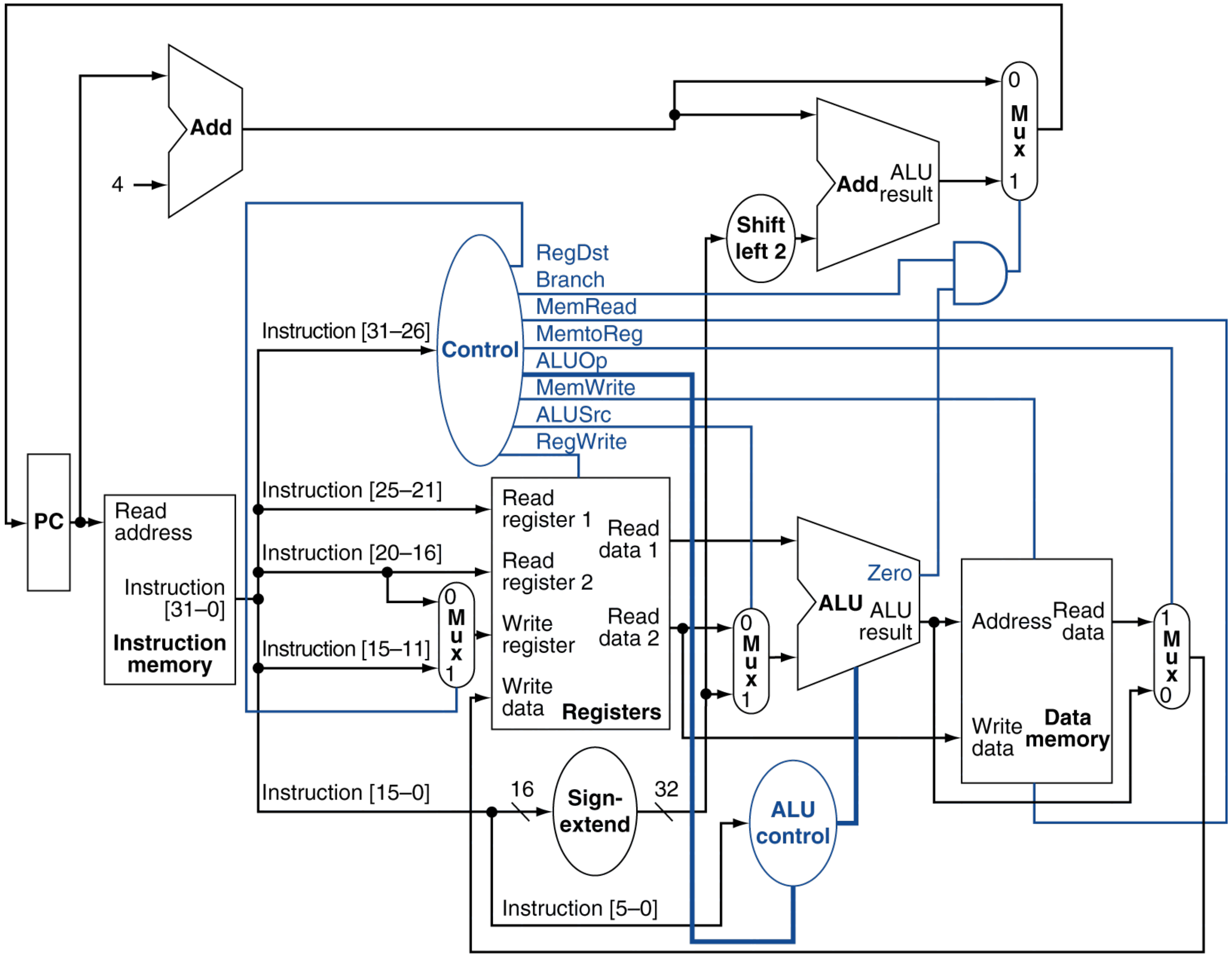
CS/EE 3810: Computer Organization

Lecture 19: Final Recap

Anton Burtsev
December
, 2022

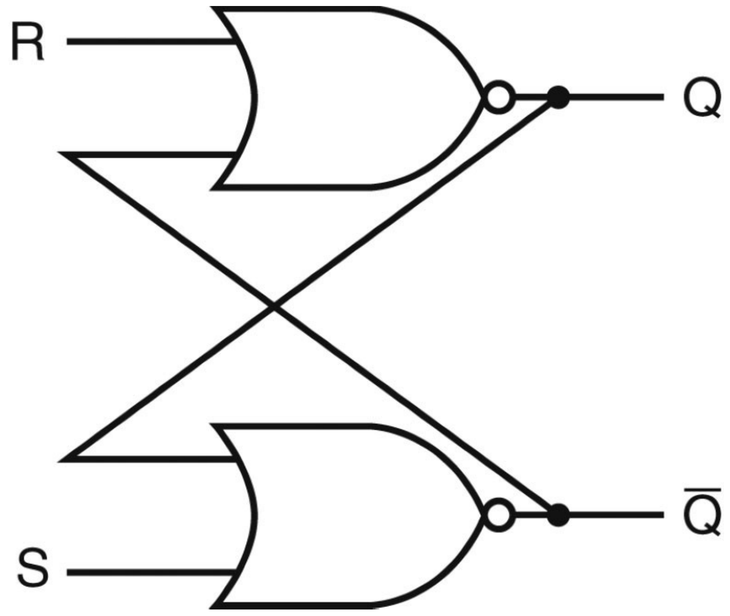
MIPS Datapath



Designing a Latch

- An S-R latch: set-reset latch
  - When Set is high, a 1 is stored
  - When Reset is high, a 0 is stored
  - When both are low, the previous state is preserved (hence, known as a storage or memory element)
  - Both are high – this set of inputs is not allowed

Verify the above behavior!

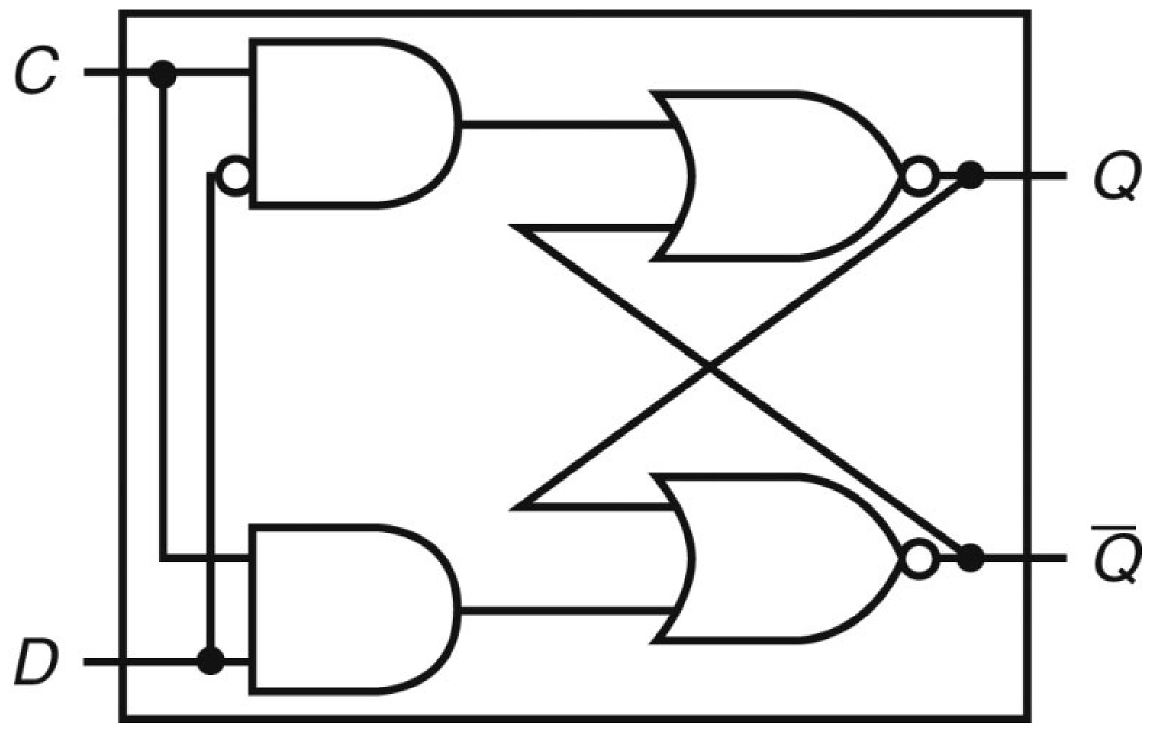


Source: H&P textbook

D Latch

- Incorporates a clock

- The value of the input D signal (data) is stored only when the clock is high – the previous state is preserved when the clock is low

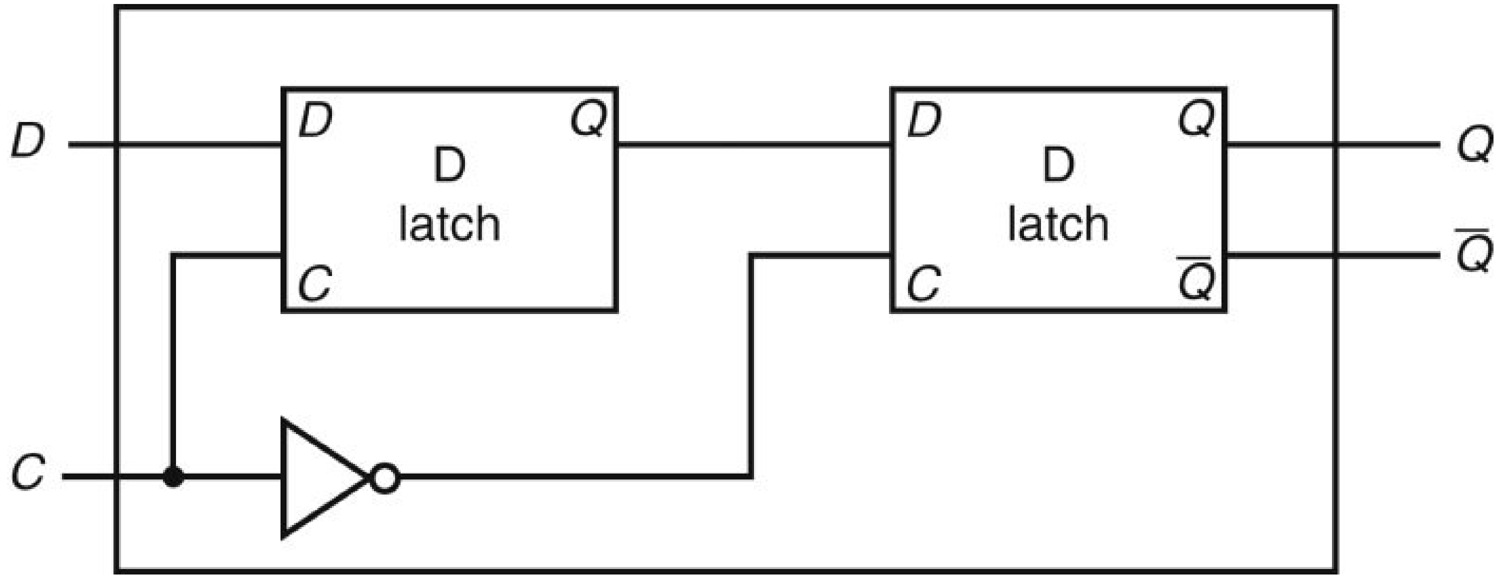


Source: H&P textbook

D Flip Flop

- Terminology:
  Latch: outputs can change any time the clock is high (asserted)
  Flip flop: outputs can change only on a clock edge

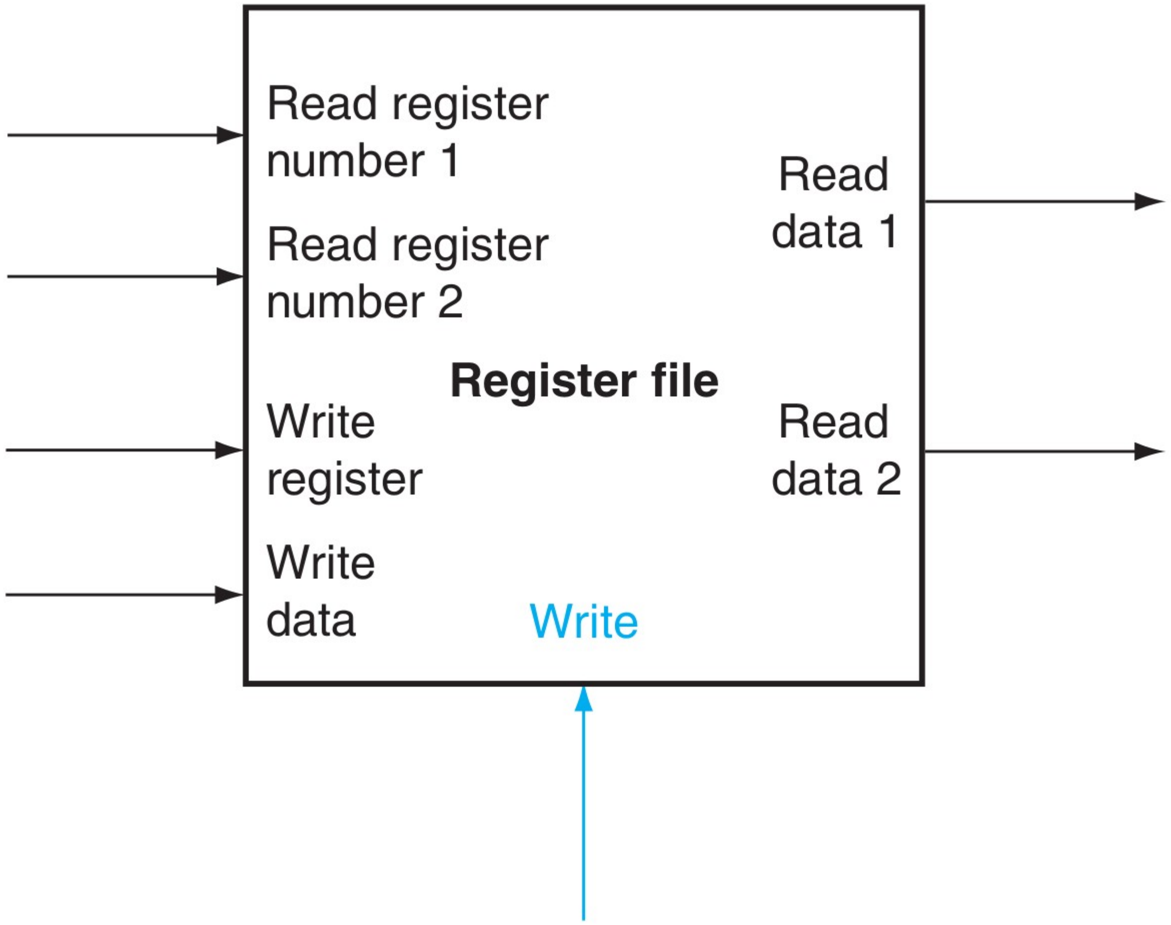
- Two D latches in series – ensures that a value is stored only on the falling edge of the clock



Source: H&P textbook

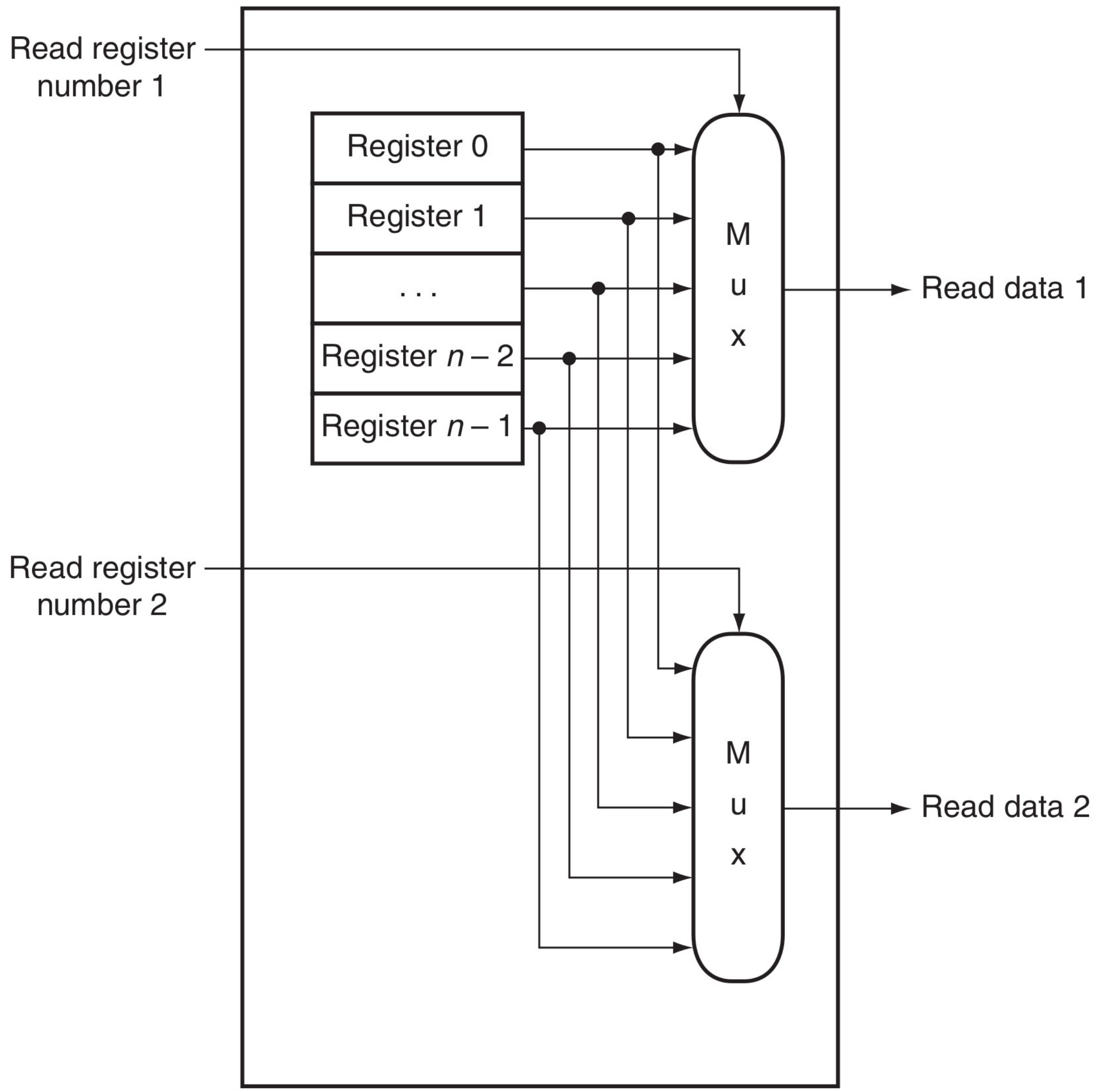
Register file

- Two read ports
- One write port



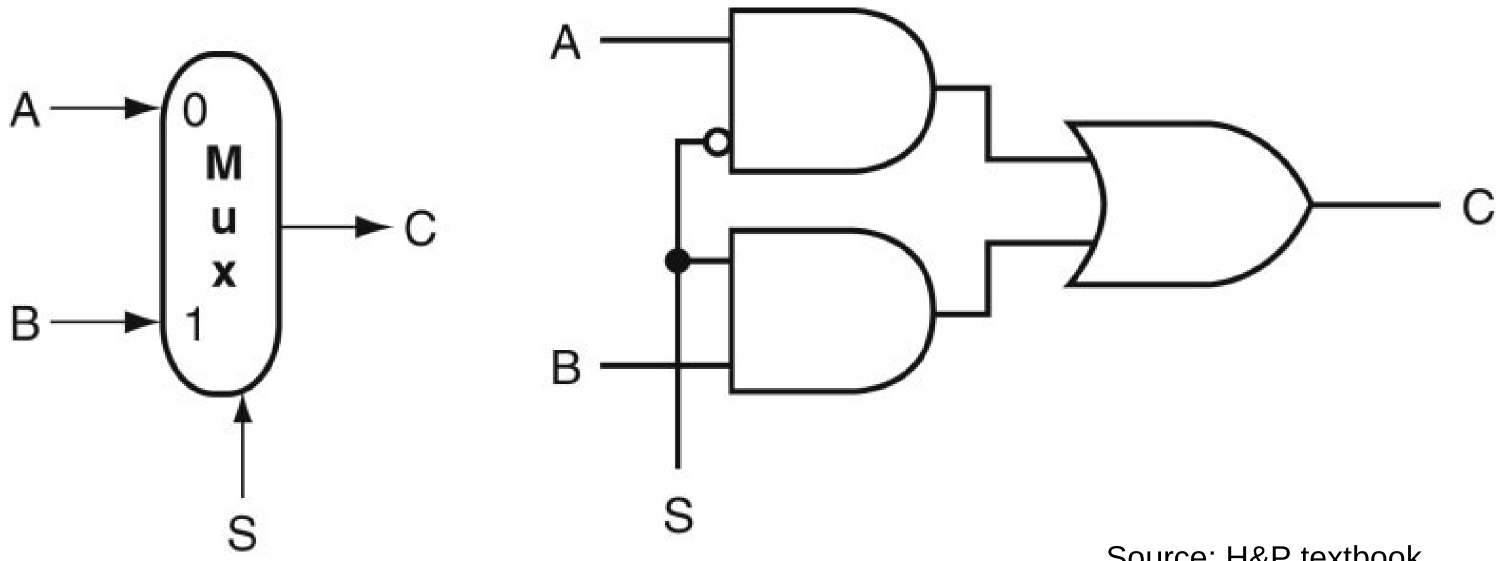
Register file (read ports)

- Build from D-flops
- Read ports
  - Multiplexors
  - N-to-1, 32bit wide



Common Logic Blocks – Multiplexor

- Multiplexor or selector: one of N inputs is reflected on the output depending on the value of the $\log_2 N$ selector bits

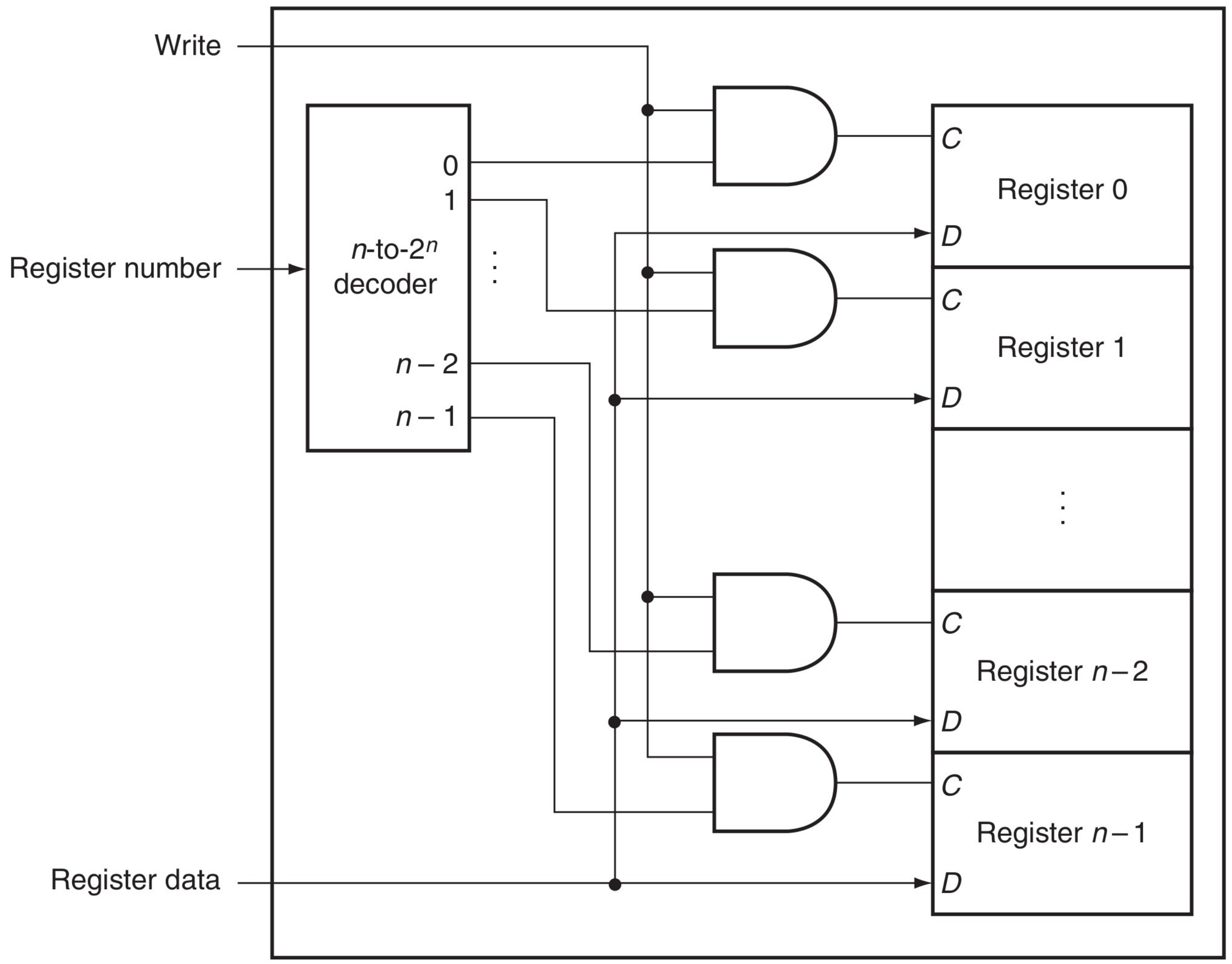


2-input mux

Source: H&P textbook

Register file (write ports)

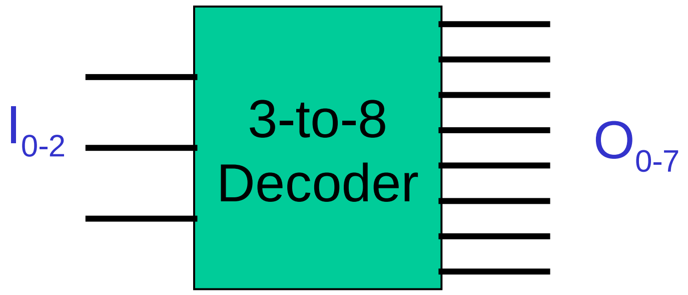
- Decoder



Common Logic Blocks – Decoder

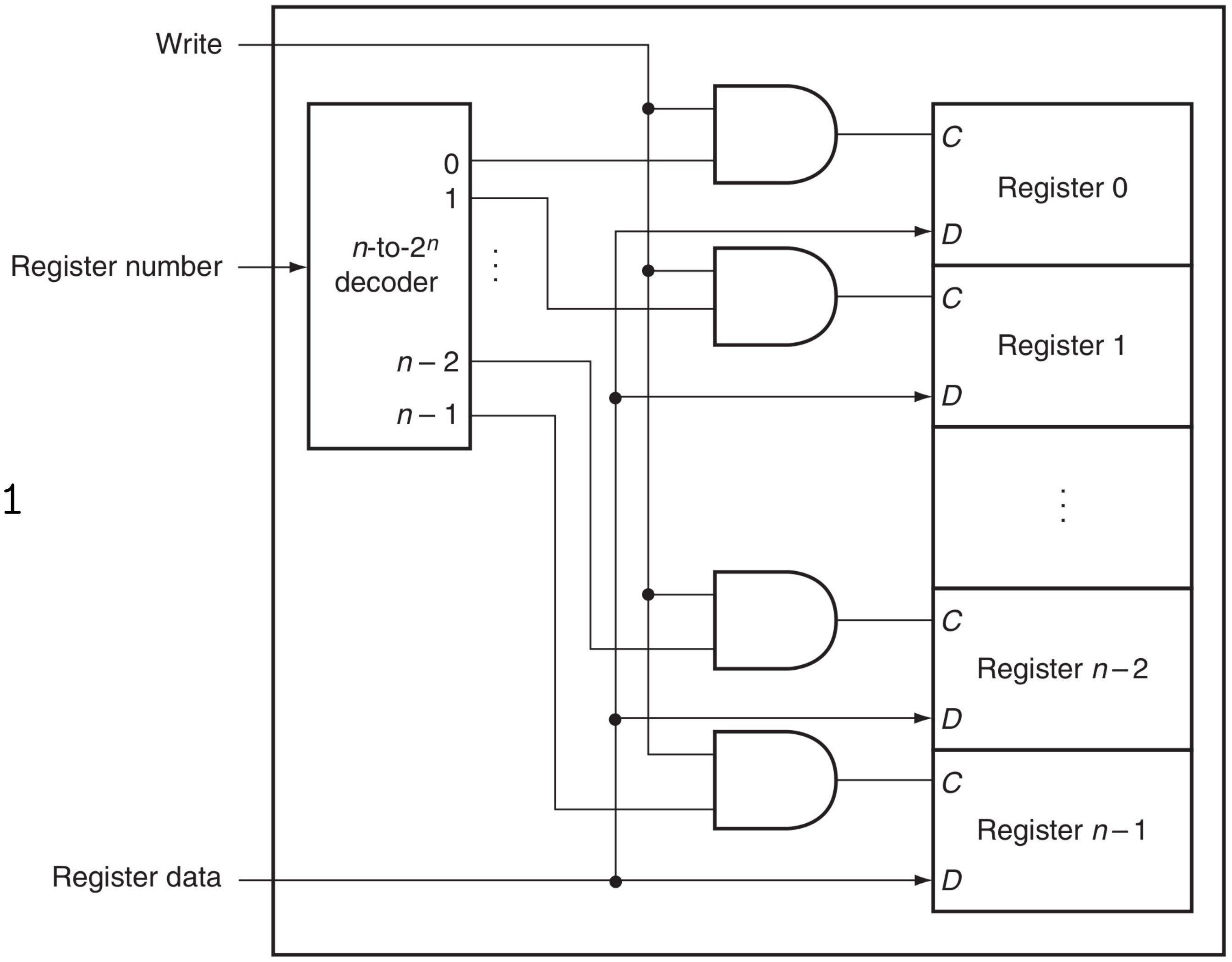
Takes in N inputs and activates one of 2^N outputs

I_0	I_1	I_2	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

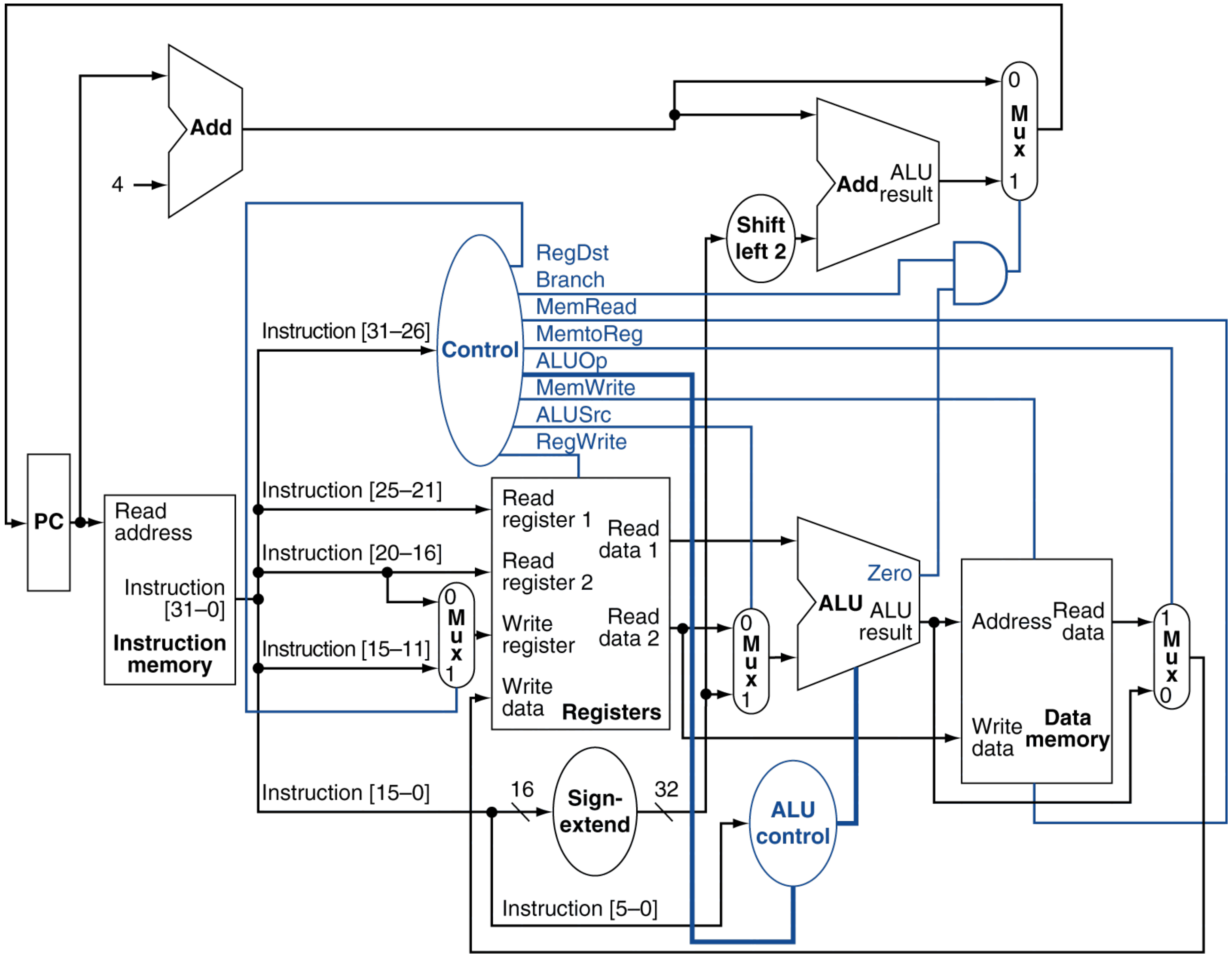


Register file (write ports)

- What if we read and write the register in the same cycle?
  - `add $s0, $s0, $s1`



MIPS Datapath



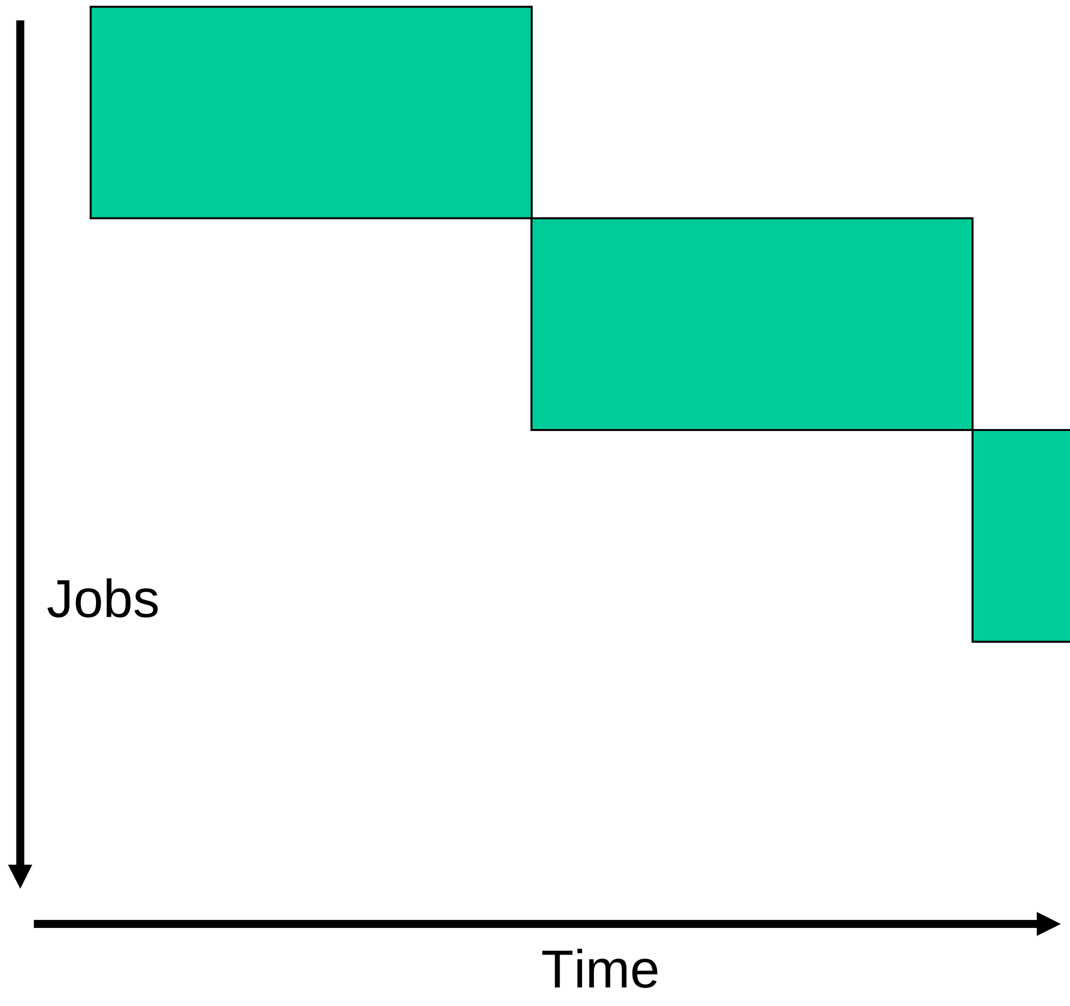
Pipelining

Building a Car

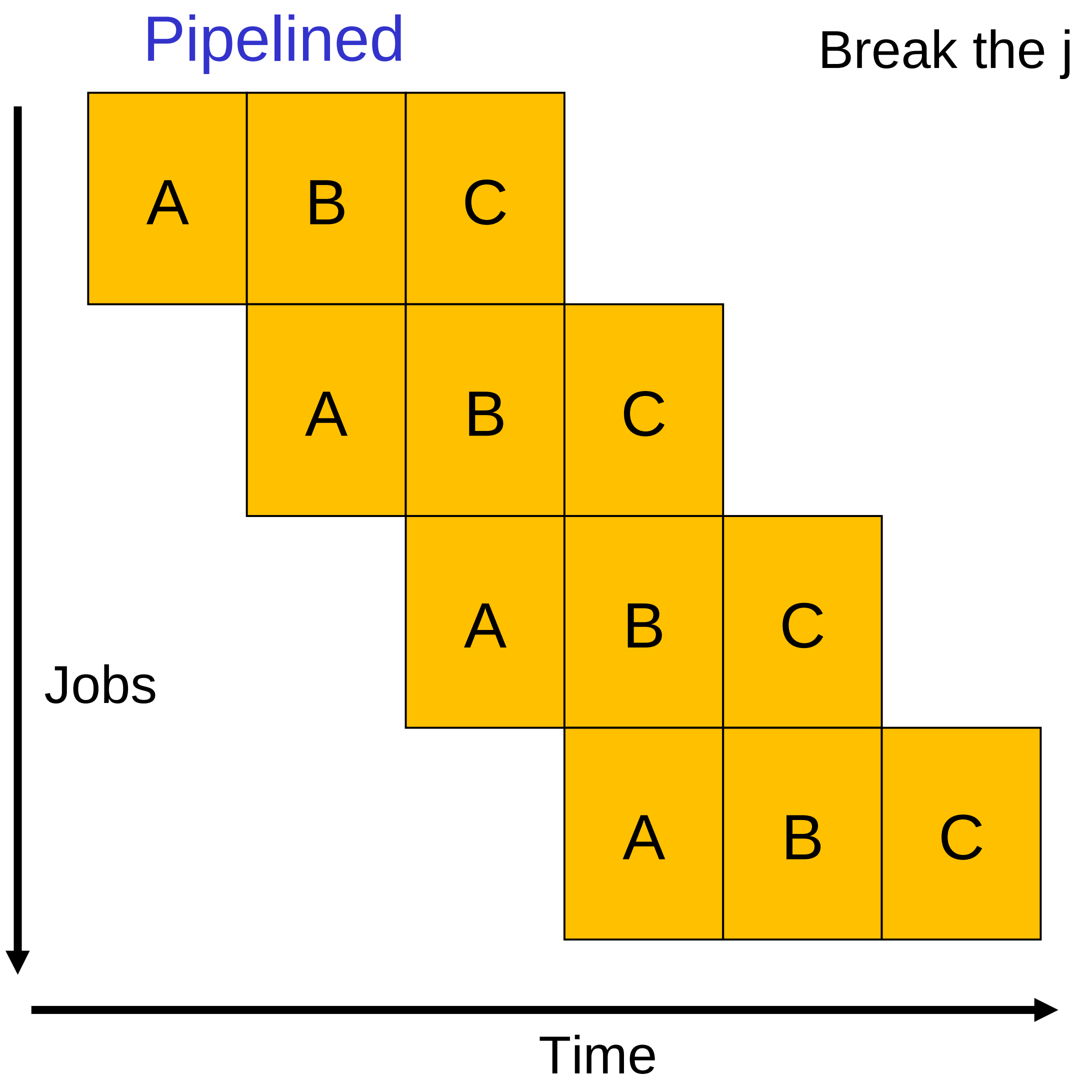
Building a Car

Unpipelined

Start and finish a job before moving to the next



The Assembly Line



Break the job into smaller stages

Performance Improvements?

Does it take longer to finish each individual job?

Does it take shorter to finish a series of jobs?

What assumptions were made while answering these questions?

Is a 10-stage pipeline better than a 5-stage pipeline?

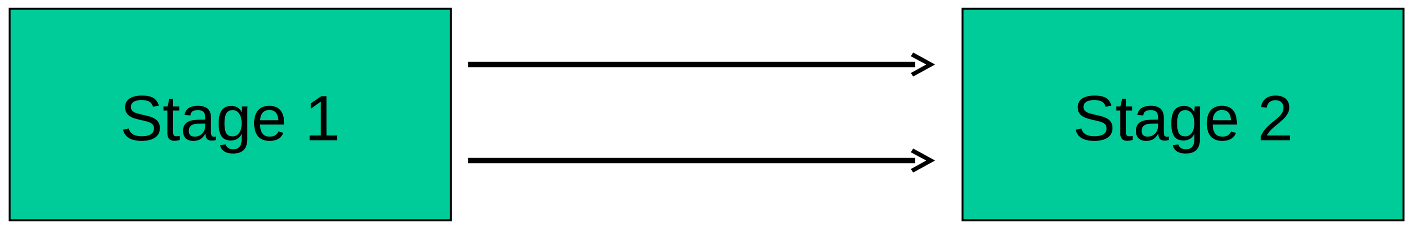
Quantitative Effects

- As a result of pipelining:
  - Time in ns per instruction goes up
  - Each instruction takes more cycles to execute
  - But… average CPI remains roughly the same
  - Clock speed goes up
  - Total execution time goes down, resulting in lower average time per instruction
  - Under ideal conditions, speedup

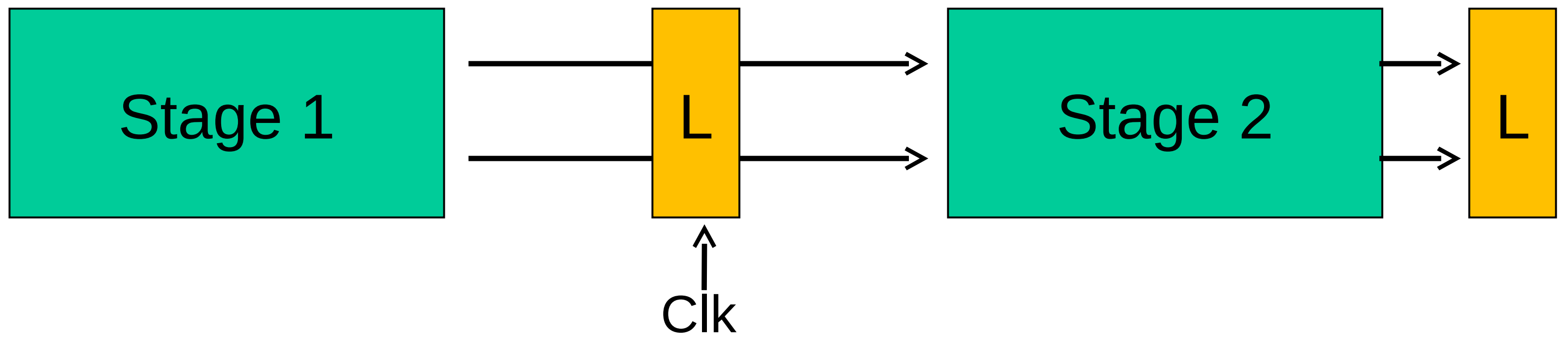
    = ratio of elapsed times between successive instruction completions

    = number of pipeline stages = increase in clock speed

Clocks and Latches



Clocks and Latches

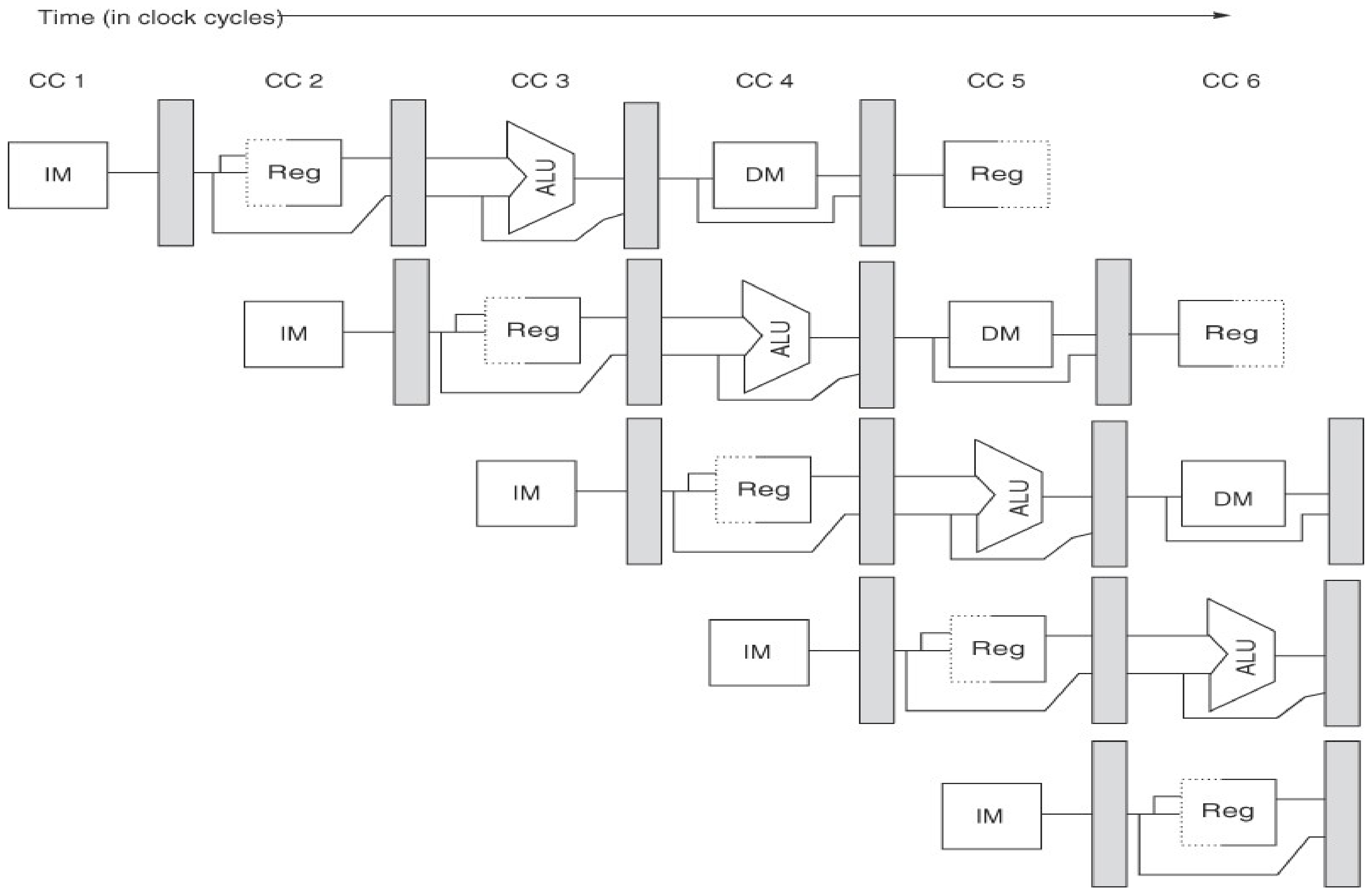


Some Equations

- Unpipelined: time to execute one instruction = $T + T_{ovh}$
- For an N-stage pipeline, time per stage = $T/N + T_{ovh}$
- Total time per instruction = $N (T/N + T_{ovh}) = T + N\ T_{ovh}$
- Clock cycle time = $T/N + T_{ovh}$
- Clock speed = $1 / (T/N + T_{ovh})$
- Ideal speedup = $(T + T_{ovh}) / (T/N + T_{ovh})$
- Cycles to complete one instruction = N
- Average CPI (cycles per instr) = 1

Hazards

A 5-Stage Pipeline



Source: H&P textbook

Hazards

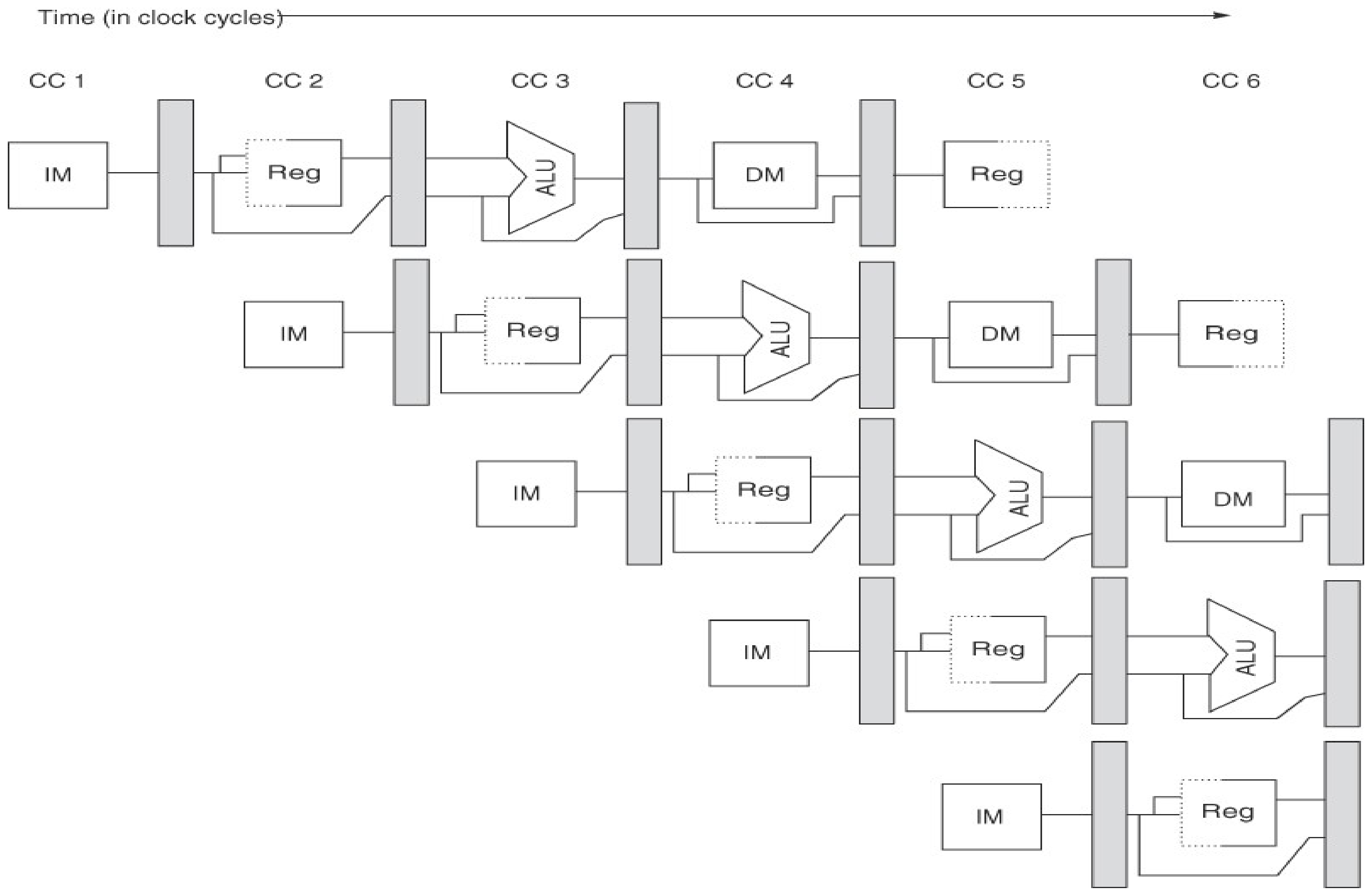
- Structural hazards: different instructions in different stages (or the same stage) conflicting for the same resource

- Data hazards: an instruction cannot continue because it needs a value that has not yet been generated by an earlier instruction

- Control hazard: fetch cannot continue because it does not know the outcome of an earlier branch – special case of a data hazard – separate category because they are treated in different ways

Structural hazards

A 5-Stage Pipeline



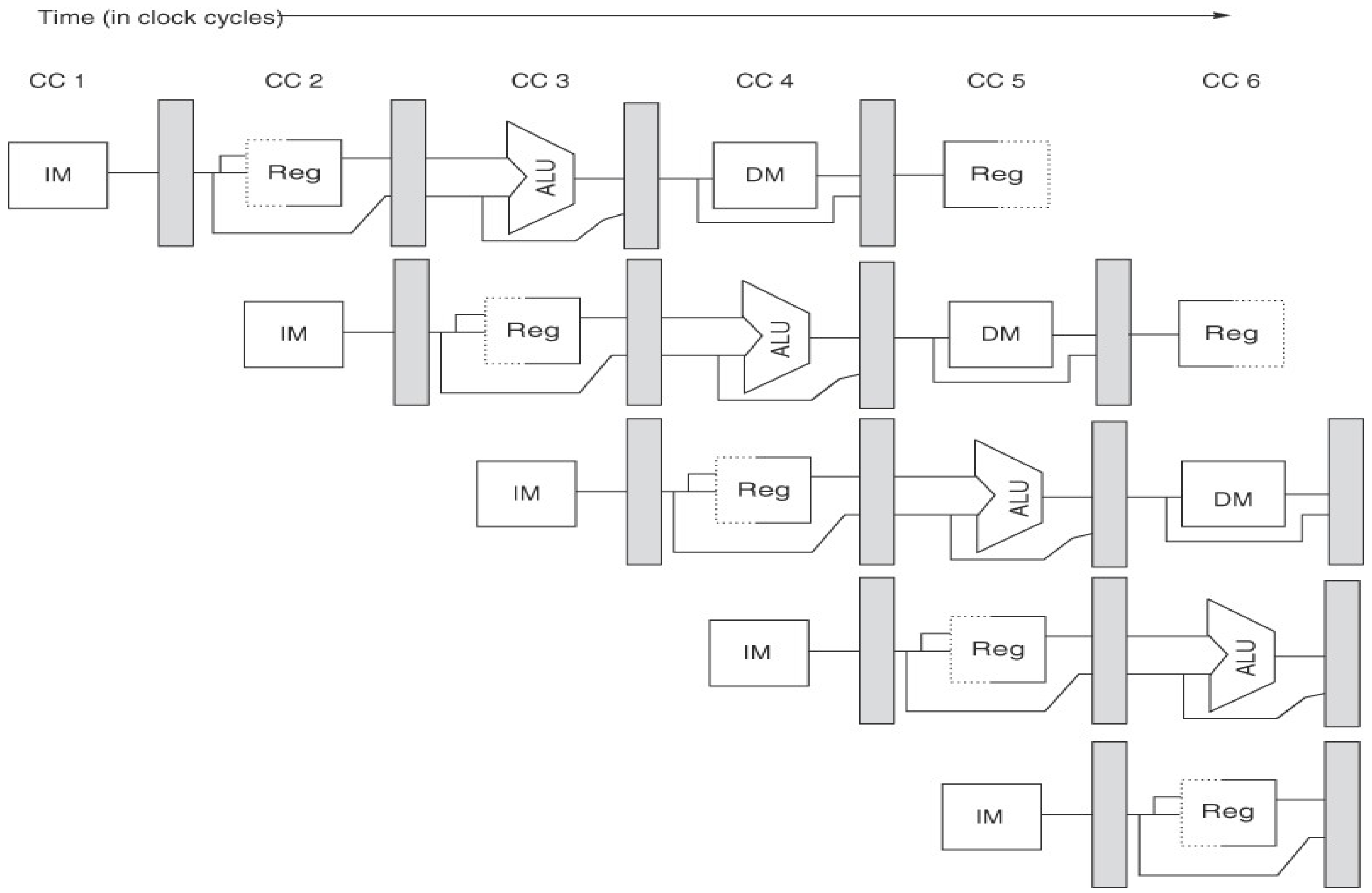
Source: H&P textbook

Structural Hazards

- Example: a unified instruction and data cache → stage 4 (MEM) and stage 1 (IF) can never coincide
- The later instruction and all its successors are delayed until a cycle is found when the resource is free → these are pipeline bubbles
- Structural hazards are easy to eliminate – increase the number of resources (for example, implement a separate instruction and data cache)

Data hazards

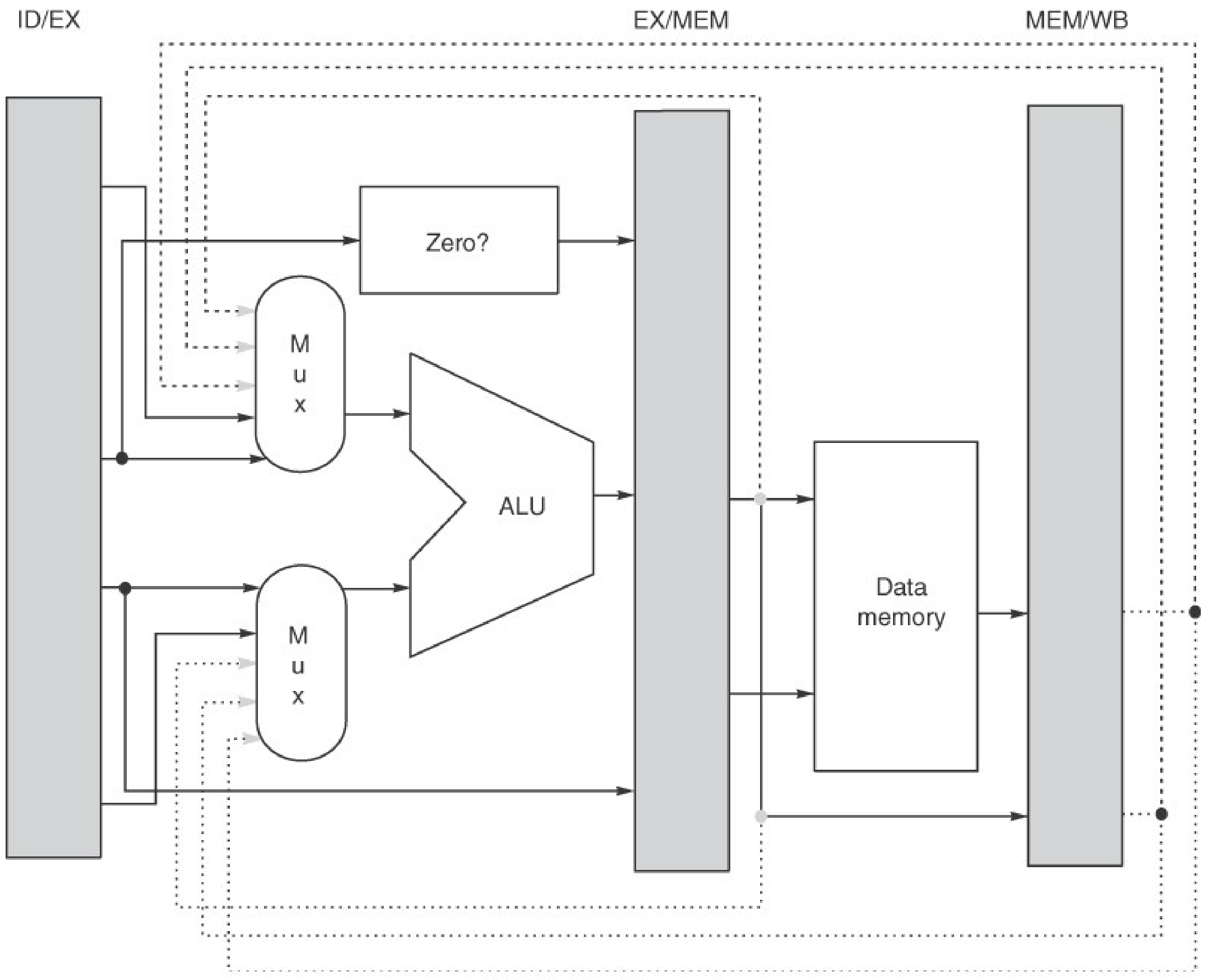
A 5-Stage Pipeline



Source: H&P textbook

Pipeline Implementation

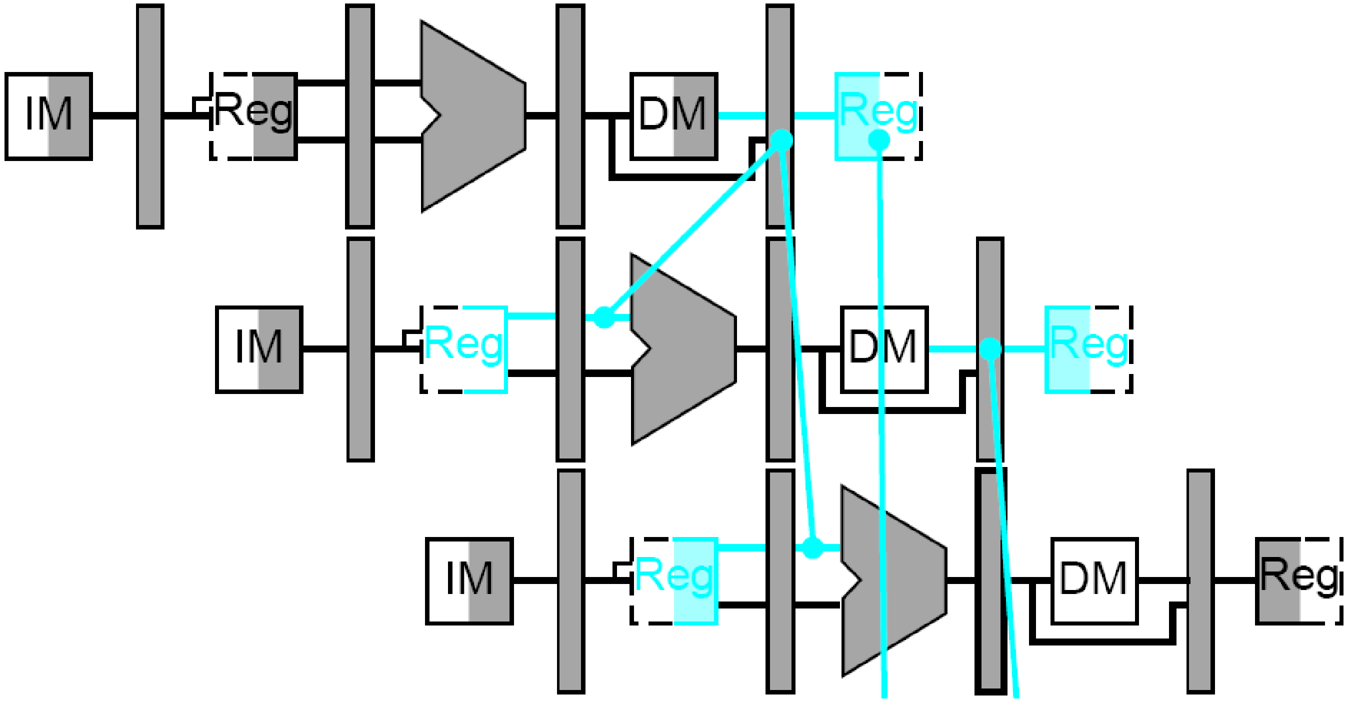
- Signals for the muxes have to be generated – some of this can happen during ID
- Need look-up tables to identify situations that merit bypassing/stalling – the number of inputs to the muxes goes up



Example

add   R1, R2, R3

lw   R4, 8(R1)

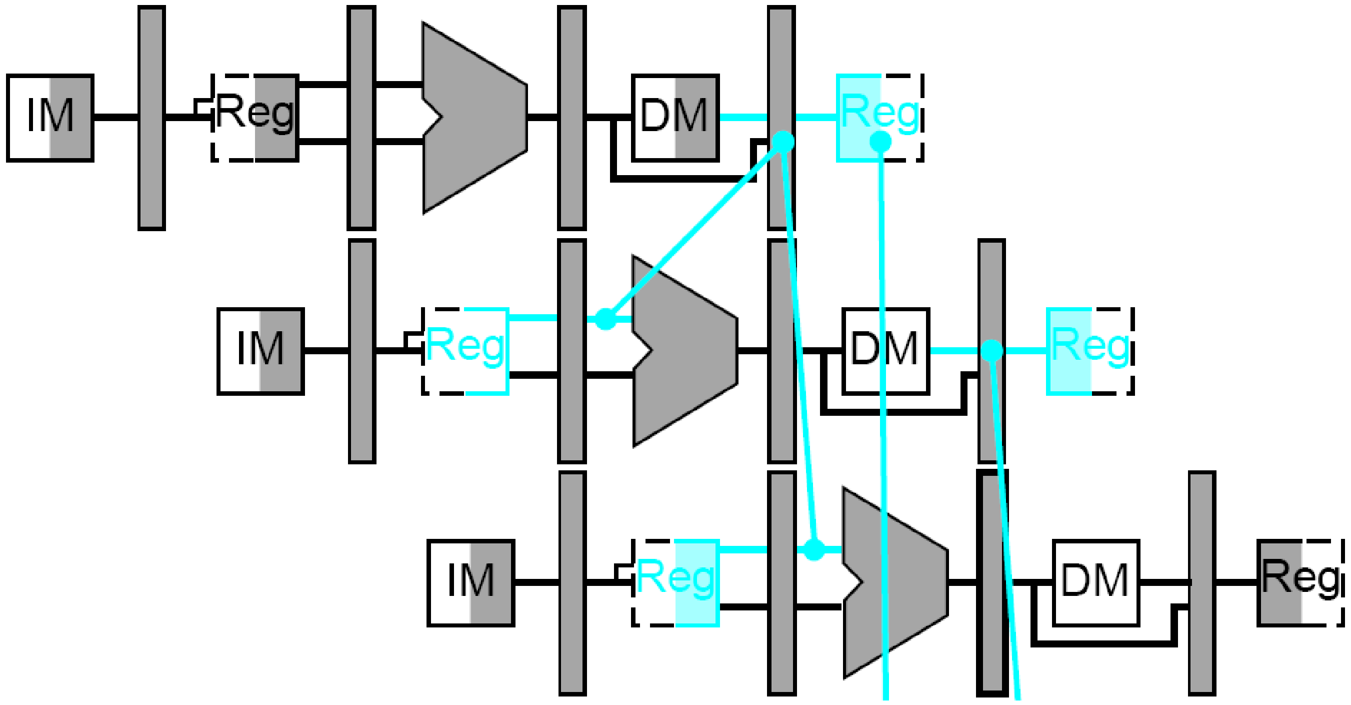


Source: H&P textbook

Example

lw R1, 8(R2)

lw R4, 8(R1)

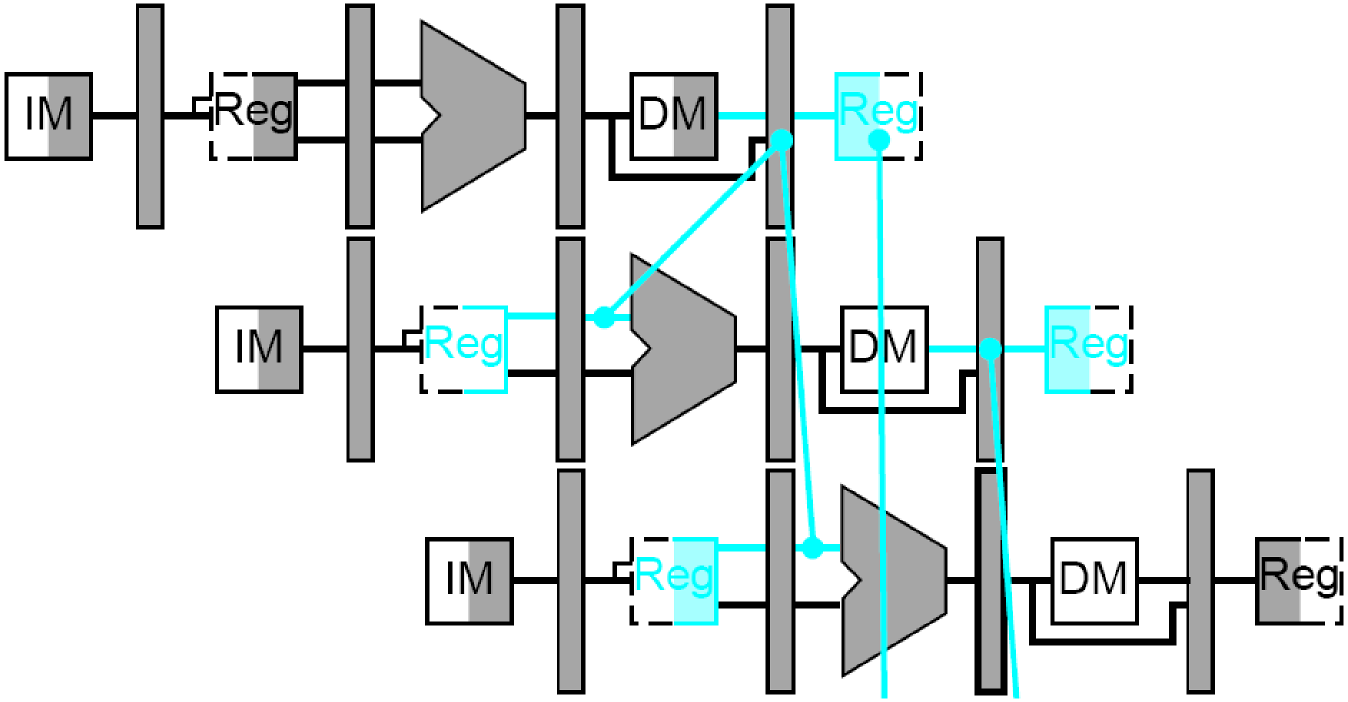


Source: H&P textbook

Example

lw   R1, 8(R2)

sw   R1, 8(R3)



Source: H&P textbook

Summary

- For the 5-stage pipeline, bypassing can eliminate delays between the following example pairs of instructions:

```
add/sub          R1, R2, R3
add/sub/lw/sw    R4, R1, R5

lw      R1, 8(R2)
sw      R1, 4(R3)
```

- The following pairs of instructions will have intermediate stalls:

```
lw             R1, 8(R2)
add/sub/lw     R3, R1, R4      or   sw   R3, 8(R1)

fmul     F1, F2, F3
fadd     F5, F1, F4
```

Control hazards

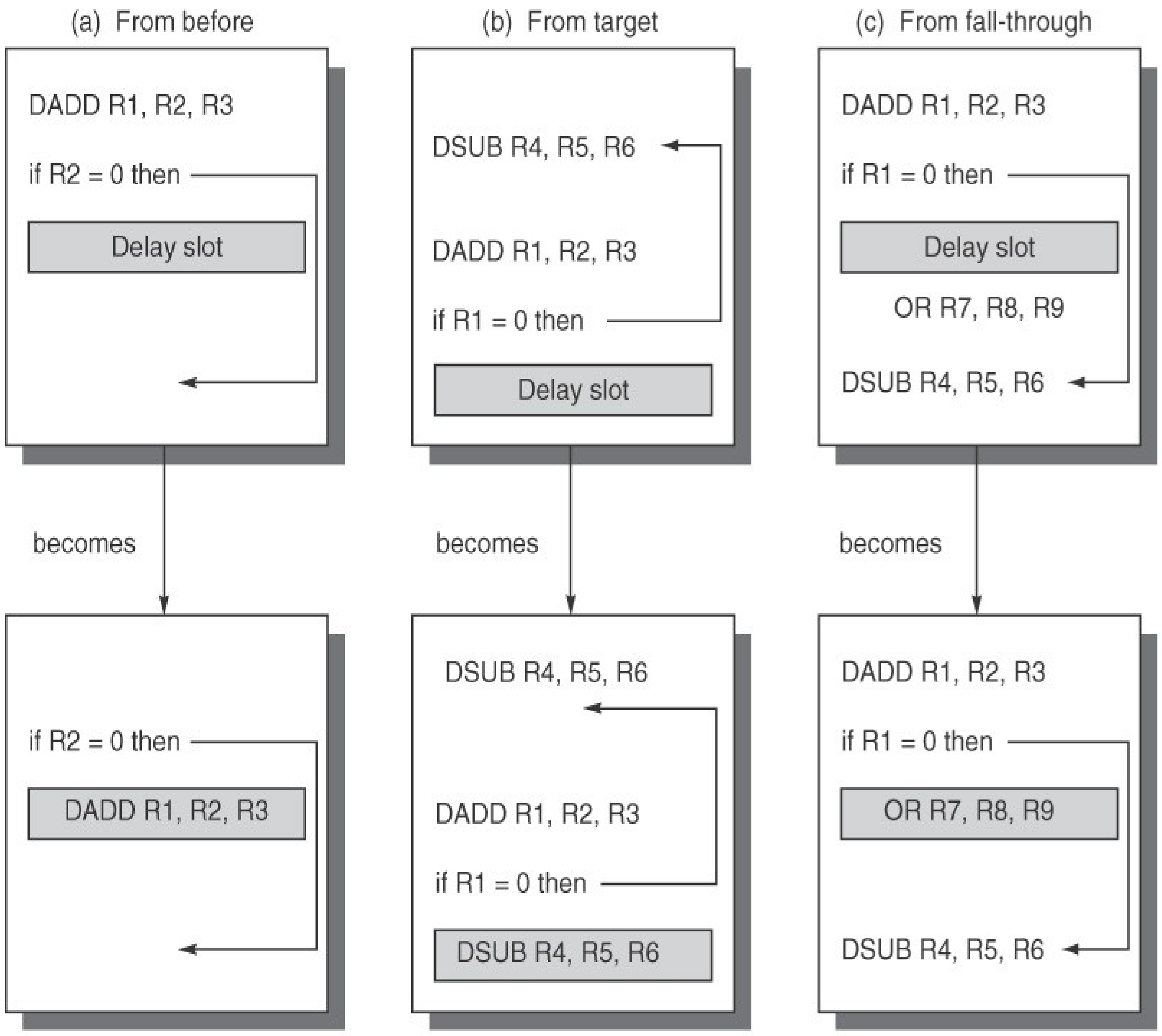
Hazards

- Structural hazards
- Data hazards
- Control hazards

Control Hazards

- Simple techniques to handle control hazard stalls:
  - for every branch, introduce a stall cycle (note: every 6th instruction is a branch on average!)
  - assume the branch is not taken and start fetching the next instruction – if the branch is taken, need hardware to cancel the effect of the wrong-path instructions
  - predict the next PC and fetch that instr – if the prediction is wrong, cancel the effect of the wrong-path instructions
  - fetch the next instruction (branch delay slot) and execute it anyway – if the instruction turns out to be on the correct path, useful work was done – if the instruction turns out to be on the wrong path, hopefully program state is not lost

Branch delay slot

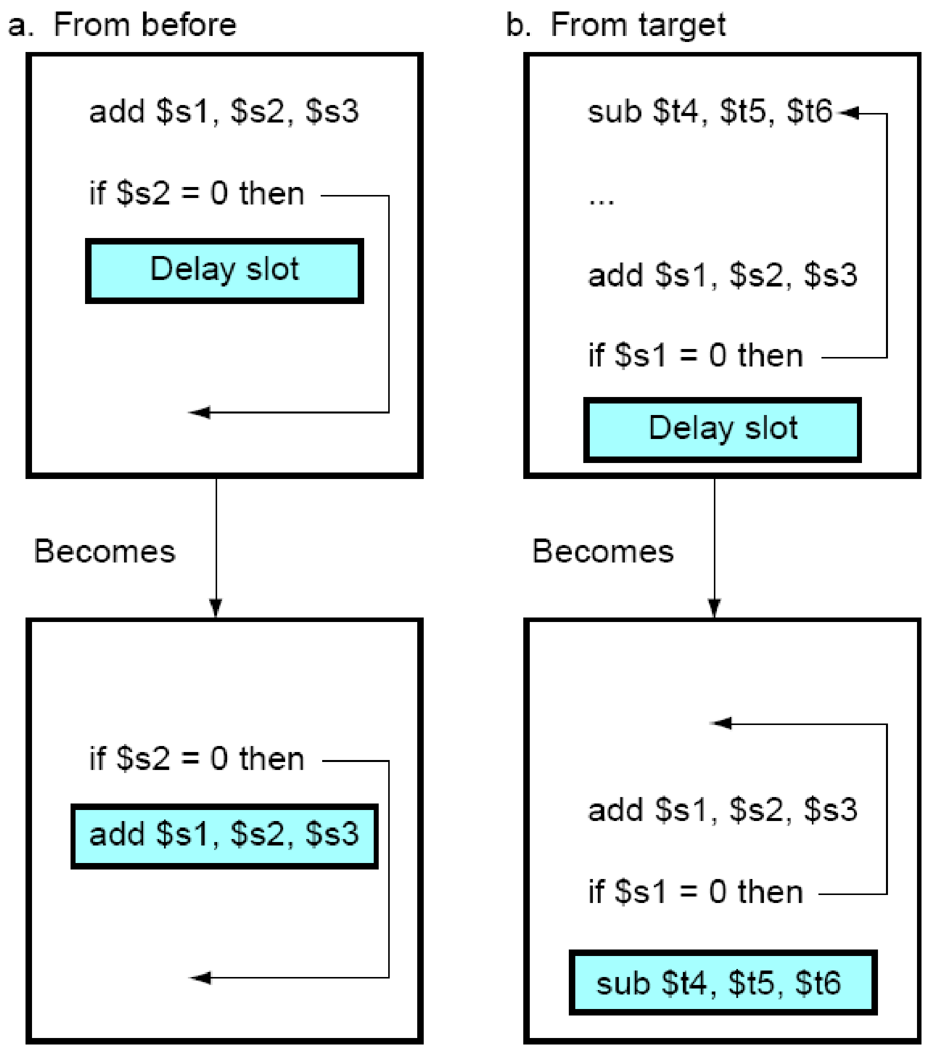


Branch prediction

Control Hazards

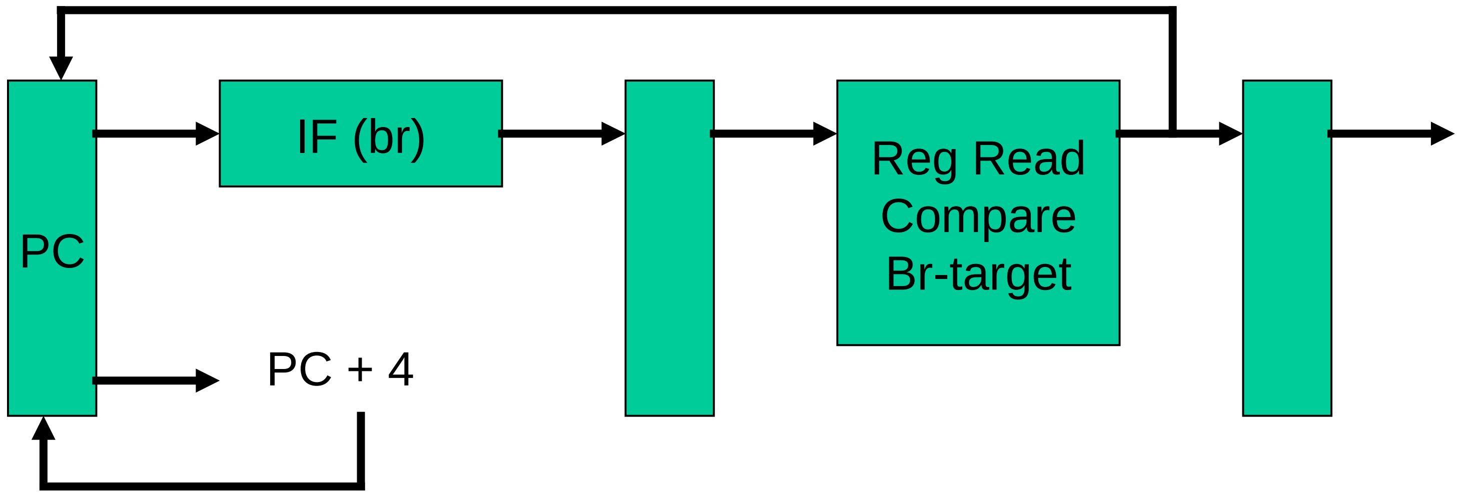
- Simple techniques to handle control hazard stalls:
  - for every branch, introduce a stall cycle (note: every 6th instruction is a branch!)
  - assume the branch is not taken and start fetching the next instruction – if the branch is taken, need hardware to cancel the effect of the wrong-path instruction
  - fetch the next instruction (branch delay slot) and execute it anyway – if the instruction turns out to be on the correct path, useful work was done – if the instruction turns out to be on the wrong path, hopefully program state is not lost
  - make a smarter guess and fetch instructions from the expected target

Branch Delay Slots



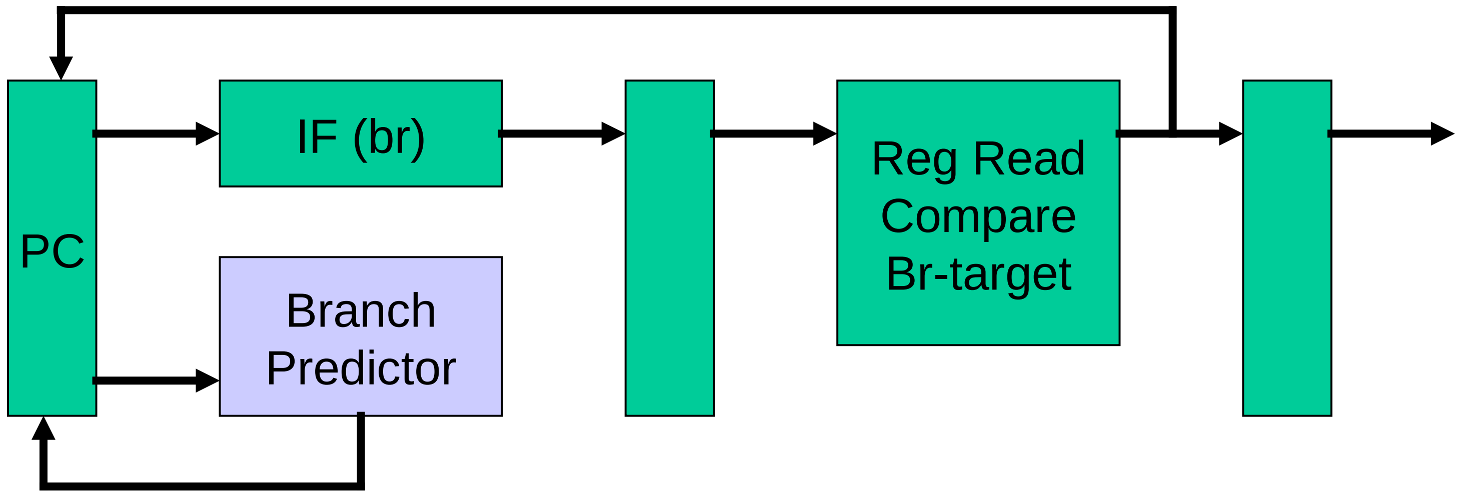
Source: H&P textbook

• Pipeline without Branch Predictor



In the 5-stage pipeline, a branch completes in two cycles →
If the branch went the wrong way, one incorrect instr is fetched →
One stall cycle per incorrect branch

• Pipeline with Branch Predictor



In the 5-stage pipeline, a branch completes in two cycles →
If the branch went the wrong way, one incorrect instr is fetched →
One stall cycle per incorrect branch

• 1-Bit Bimodal Prediction

- For each branch, keep track of what happened last time and use that outcome as the prediction
- What are prediction accuracies for branches 1 and 2 below:

```
while (1) {
    for (i=0;i<10;i++) {                branch-1
        …
    }
    for (j=0;j<20;j++) {                branch-2
        …
    }
}
```

• 2-Bit Bimodal Prediction

- For each branch, maintain a 2-bit saturating counter:
  if the branch is taken: counter = min(3,counter+1)
  if the branch is not taken: counter = max(0,counter-1)

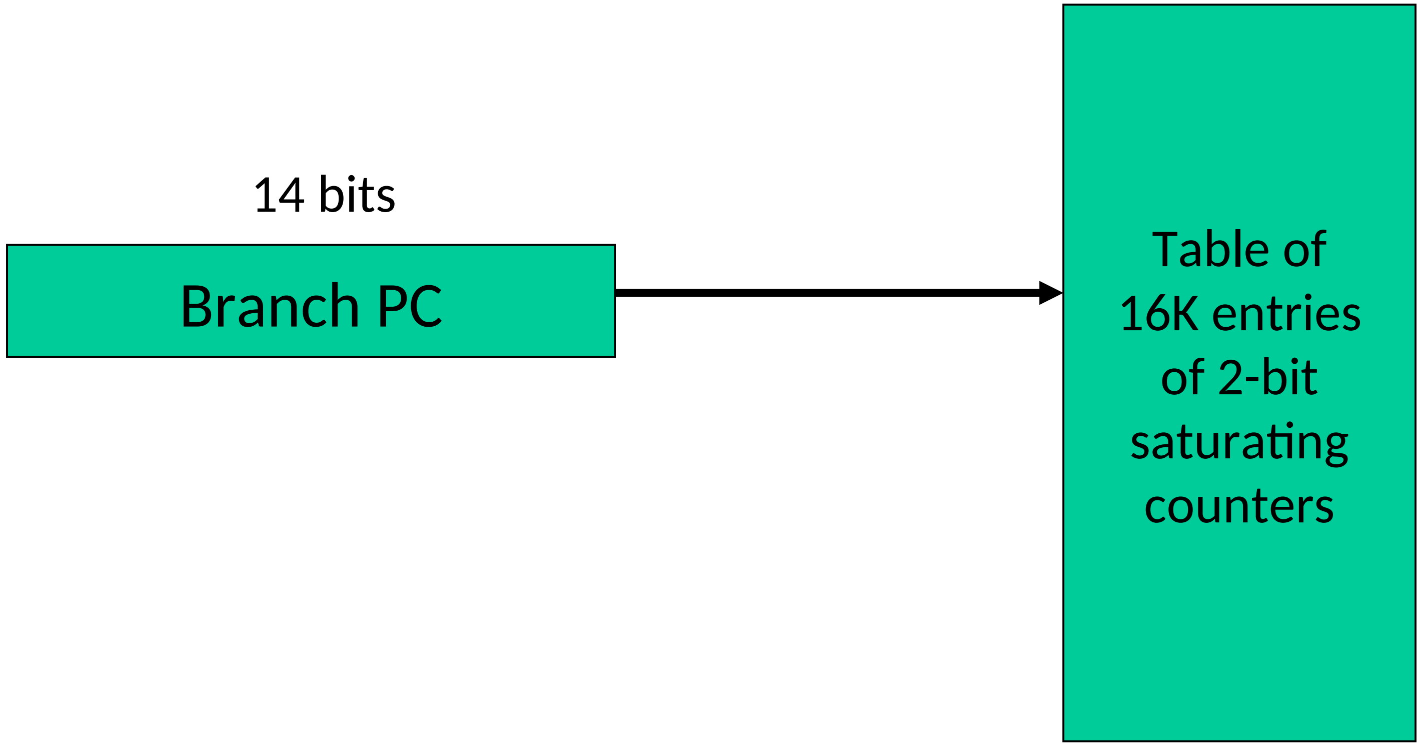
- If (counter >= 2), predict taken, else predict not taken

- Advantage: a few atypical branches will not influence the prediction (a better measure of "the common case")

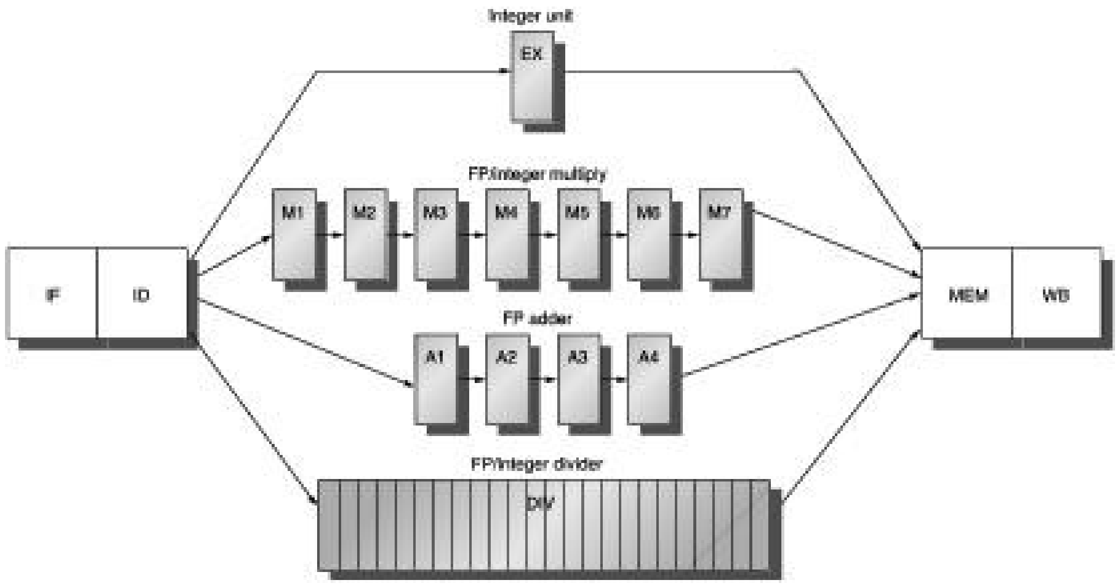
- Especially useful when multiple branches share the same counter (some bits of the branch PC are used to index into the branch predictor)

- Can be easily extended to N-bits (in most processors, N=2)

Bimodal Predictor



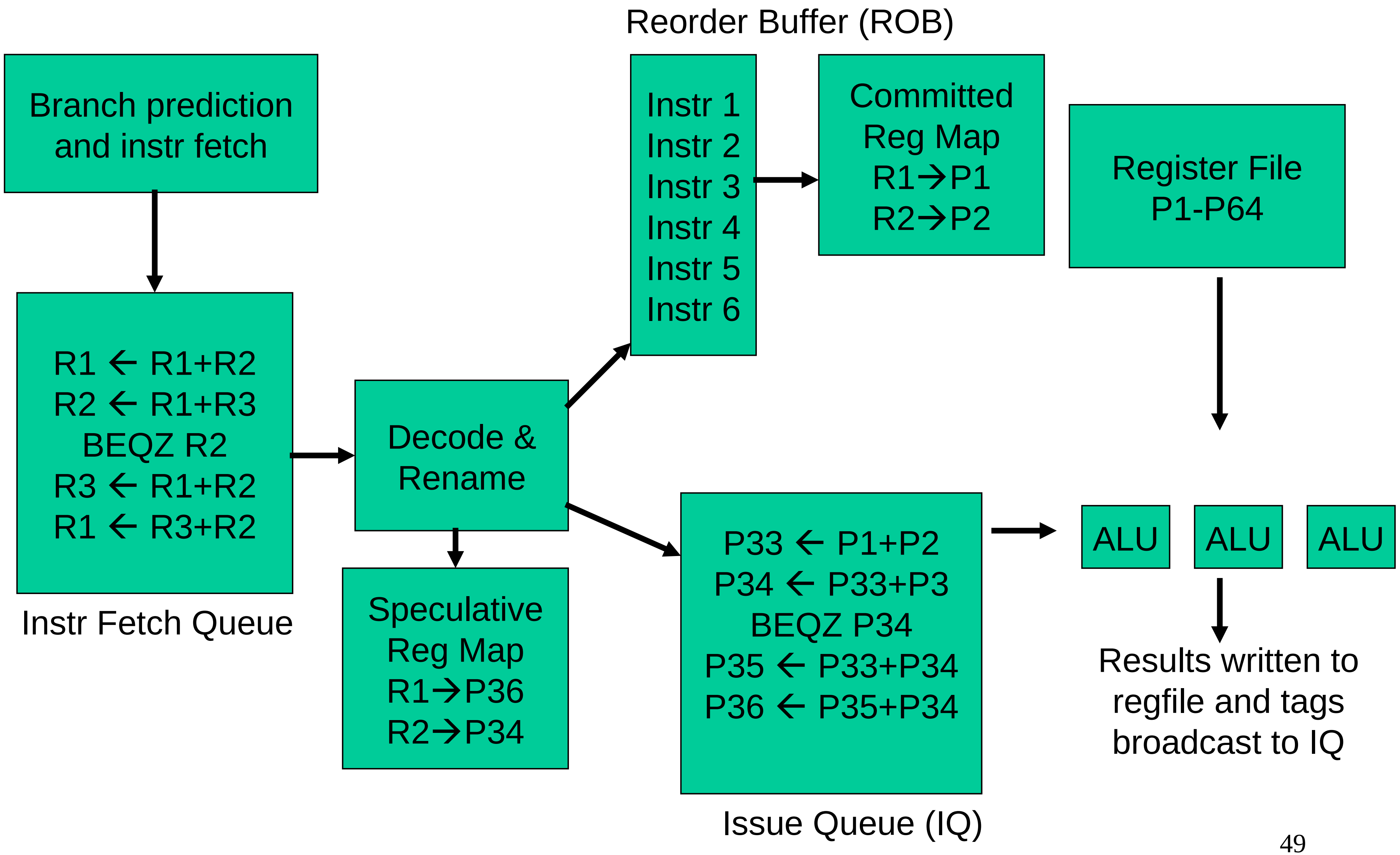
Multicycle Instructions



- Multiple parallel pipelines – each pipeline can have a different number of stages

- Instructions can now complete out of order – must make sure that writes to a register happen in the correct order

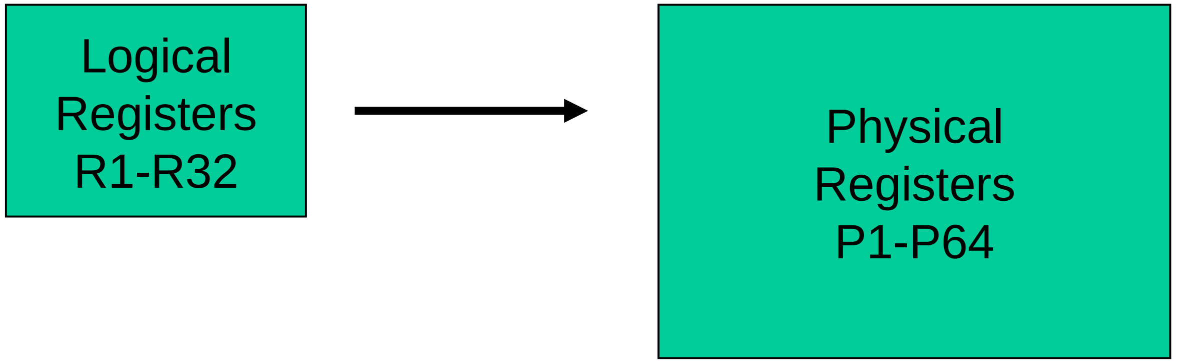
• The Alpha 21264 Out-of-Order Implementation



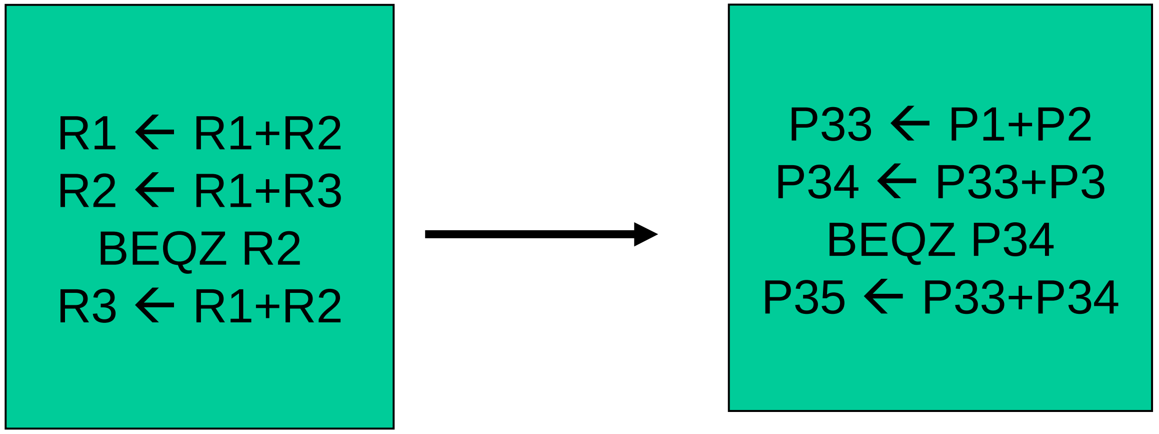
- Logical and physical registers

• Managing Register Names

Temporary values are stored in the register file and not the ROB



At the start, R1-R32 can be found in P1-P32
Instructions stop entering the pipeline when P64 is assigned



```
R1 ← R1+R2
R2 ← R1+R3
BEQZ R2
R3 ← R1+R2
```

```
P33 ← P1+P2
P34 ← P33+P3
BEQZ P34
P35 ← P33+P34
```

What happens on commit?

The Commit Process

- On commit, no copy is required
- The register map table is updated – the “committed” value of R1 is now in P33 and not P1 – on an exception, P33 is copied to memory and not P1
- An instruction in the issue queue need not modify its input operand when the producer commits
- When instruction-1 commits, we no longer have any use for P1 – it is put in a free pool and a new instruction can now enter the pipeline → for every instr that commits, a new instr can enter the pipeline → number of in-flight instrs is a constant = number of extra (rename) registers

- Additional Details

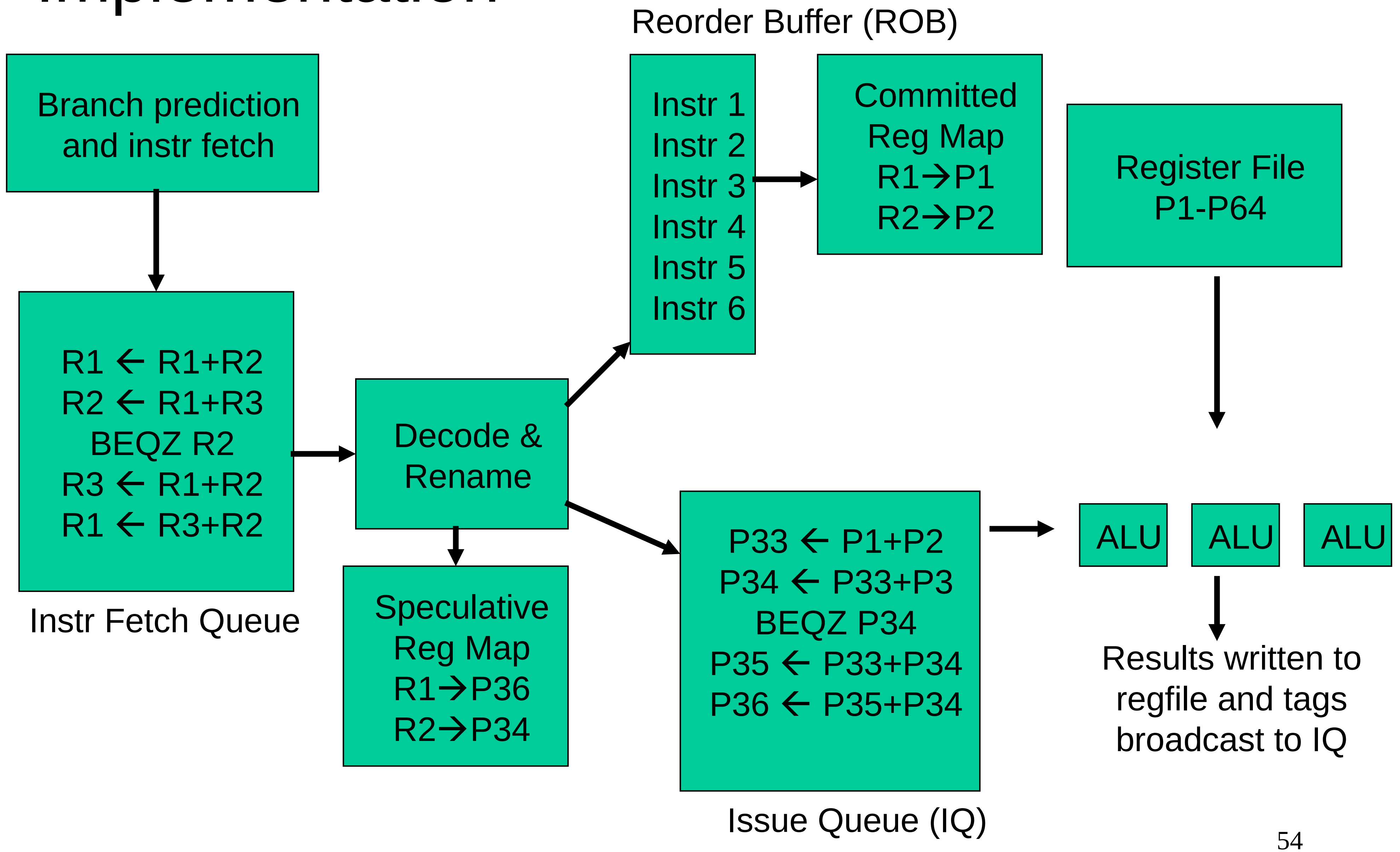
- When does the decode stage stall? When we either run out of registers, or ROB entries, or issue queue entries

- Issue width: the number of instructions handled by each stage in a cycle. High issue width ➔ high peak ILP

- Window size: the number of in-flight instructions in the pipeline. Large window size ➔ high ILP

- No more WAR and WAW hazards because of rename registers – must only worry about RAW hazards

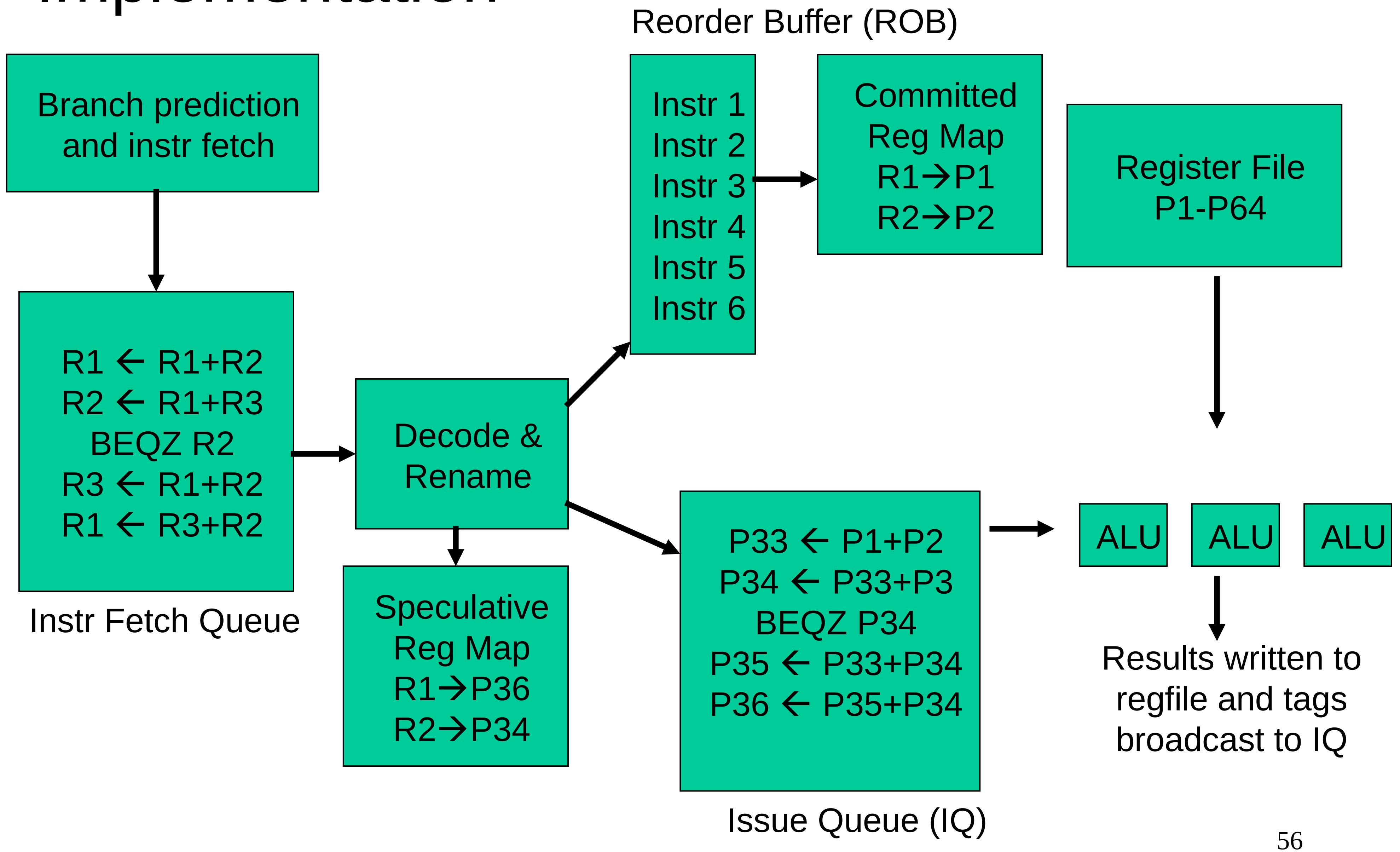
• The Alpha 21264 Out-of-Order Implementation



• Additional Details

- When does the decode stage stall? When we either run out of registers, or ROB entries, or issue queue entries
- Issue width: the number of instructions handled by each stage in a cycle. High issue width ➔ high peak ILP
- Window size: the number of in-flight instructions in the pipeline. Large window size ➔ high ILP
- No more WAR and WAW hazards because of rename registers – must only worry about RAW hazards

• The Alpha 21264 Out-of-Order Implementation



Caches

Cache Hierarchies

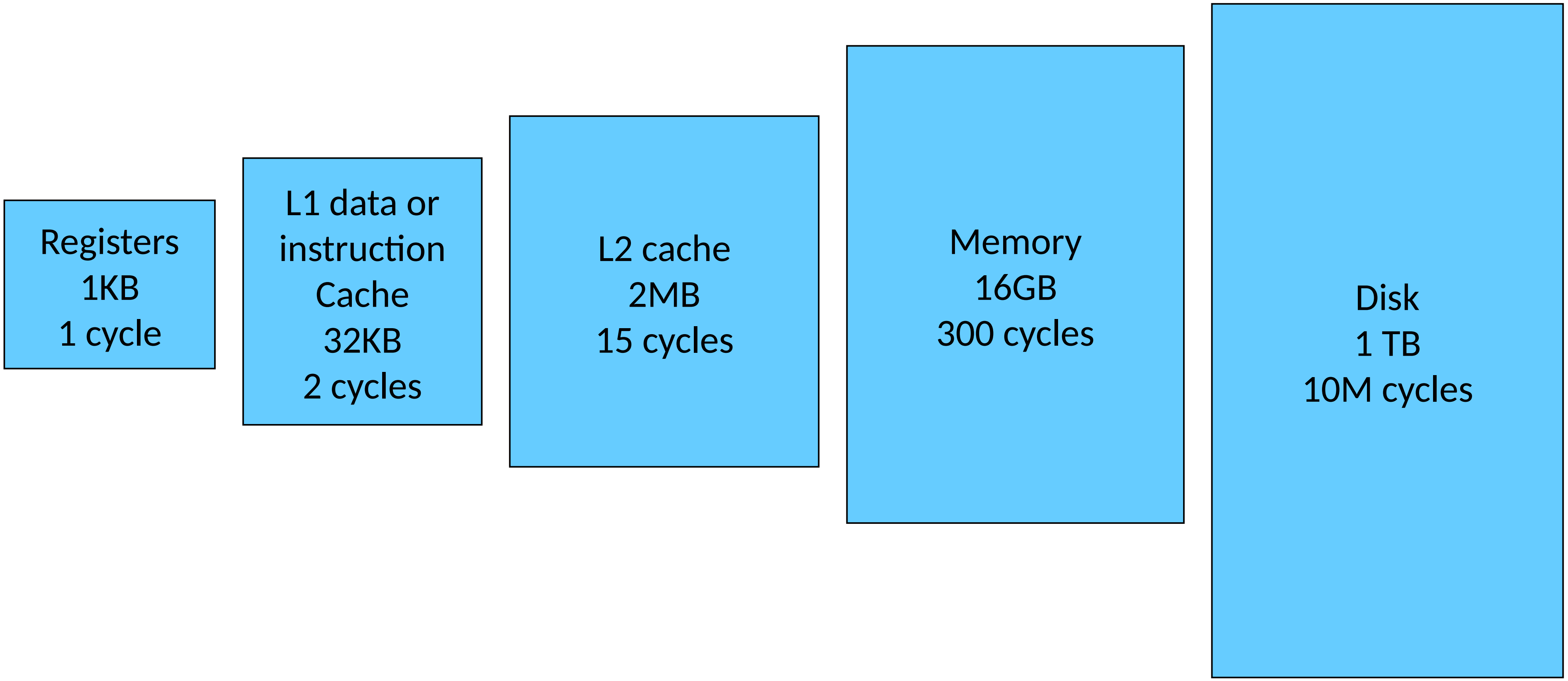
- Data and instructions are stored on DRAM chips – DRAM is a technology that has high bit density, but relatively poor latency – an access to data in memory can take as many as 300 cycles today!

- Hence, some data is stored on the processor in a structure called the cache – caches employ SRAM technology, which is faster, but has lower bit density

- Internet browsers also cache web pages – same concept

Memory Hierarchy

- As you go further, capacity and latency increase



Locality

- Why do caches work?
  - Temporal locality: if you used some data recently, you will likely use it again
  - Spatial locality: if you used some data recently, you will likely access its neighbors

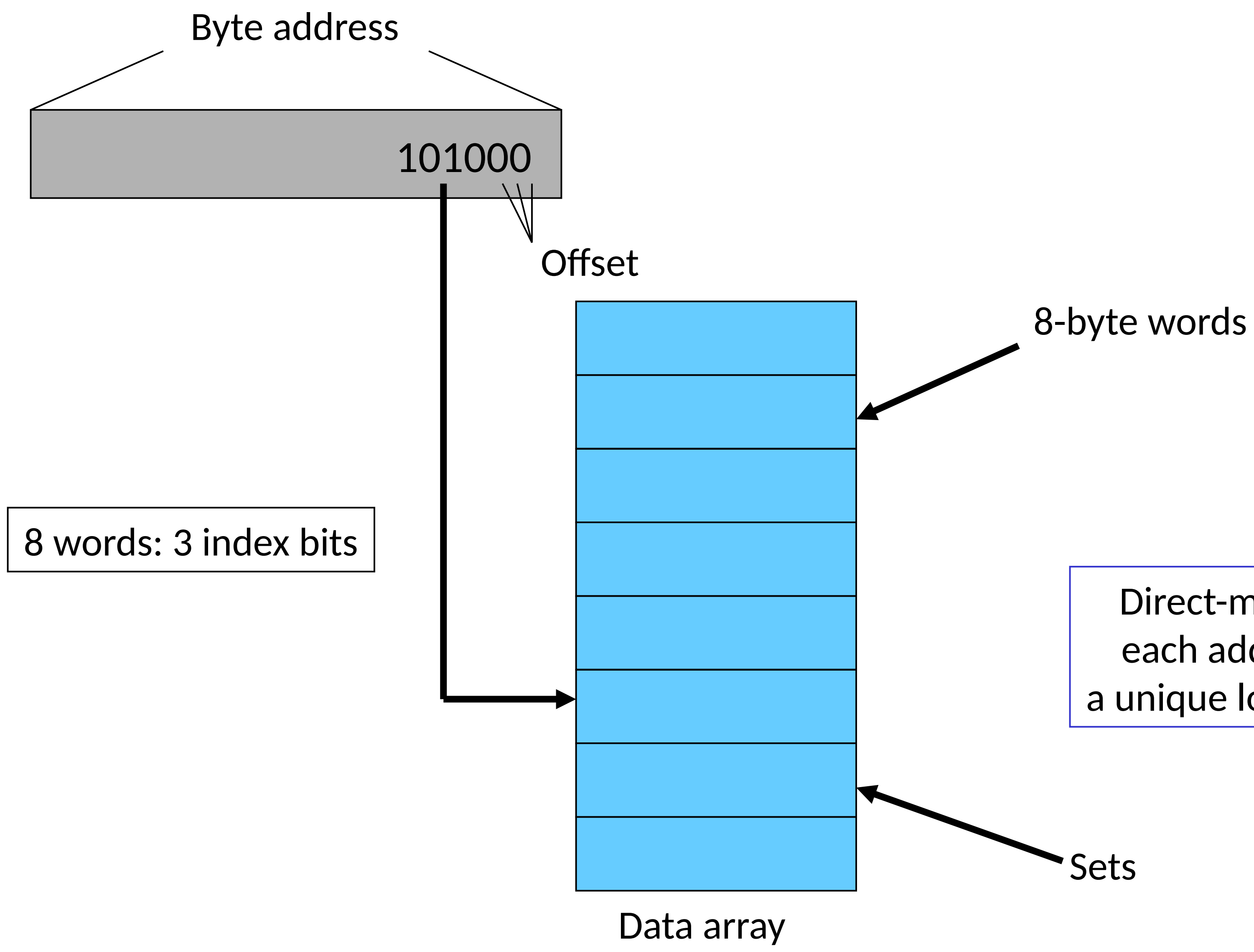
- No hierarchy: average access time for data = 300 cycles

- 32KB 1-cycle L1 cache that has a hit rate of 95%:

  average access time = 0.95 x 1 + 0.05 x (301)

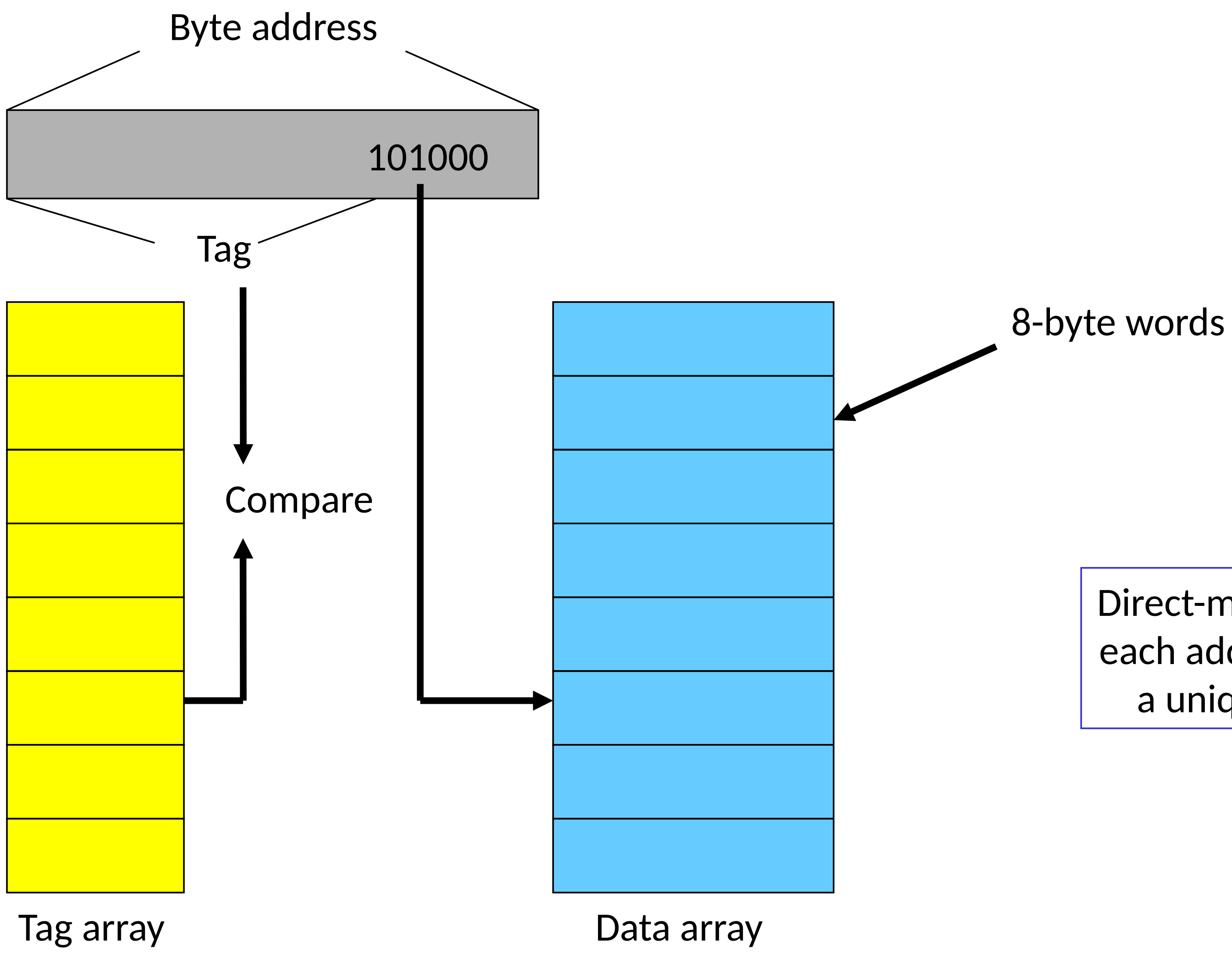
  = 16 cycles

Accessing the Cache



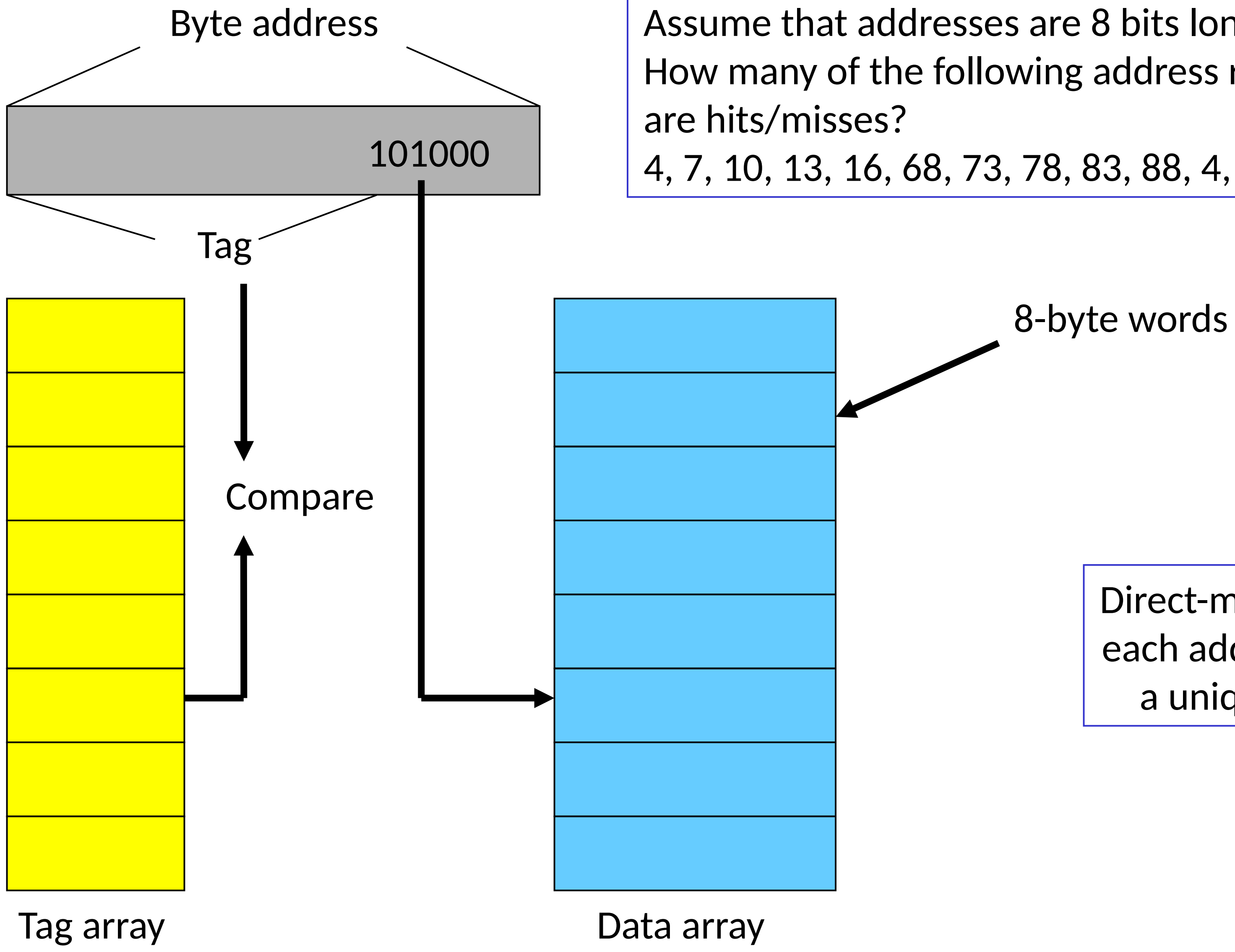
Direct-mapped cache:
each address maps to
a unique location in cache

The Tag Array



Direct-mapped cache: each address maps to a unique address

Example Access Pattern

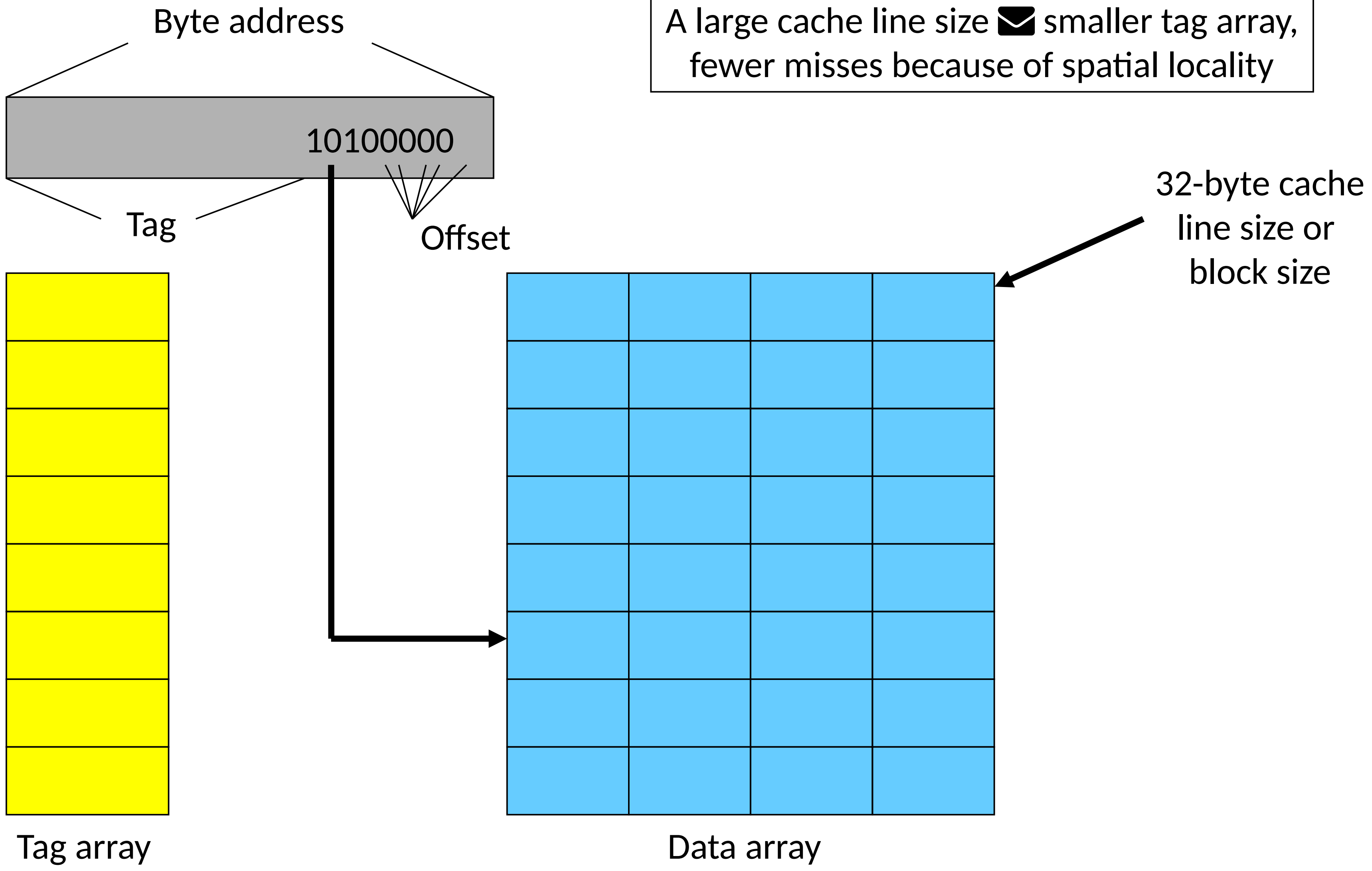


Assume that addresses are 8 bits long
How many of the following address requests are hits/misses?
4, 7, 10, 13, 16, 68, 73, 78, 83, 88, 4, 7, 10…

Direct-mapped cache: each address maps to a unique address

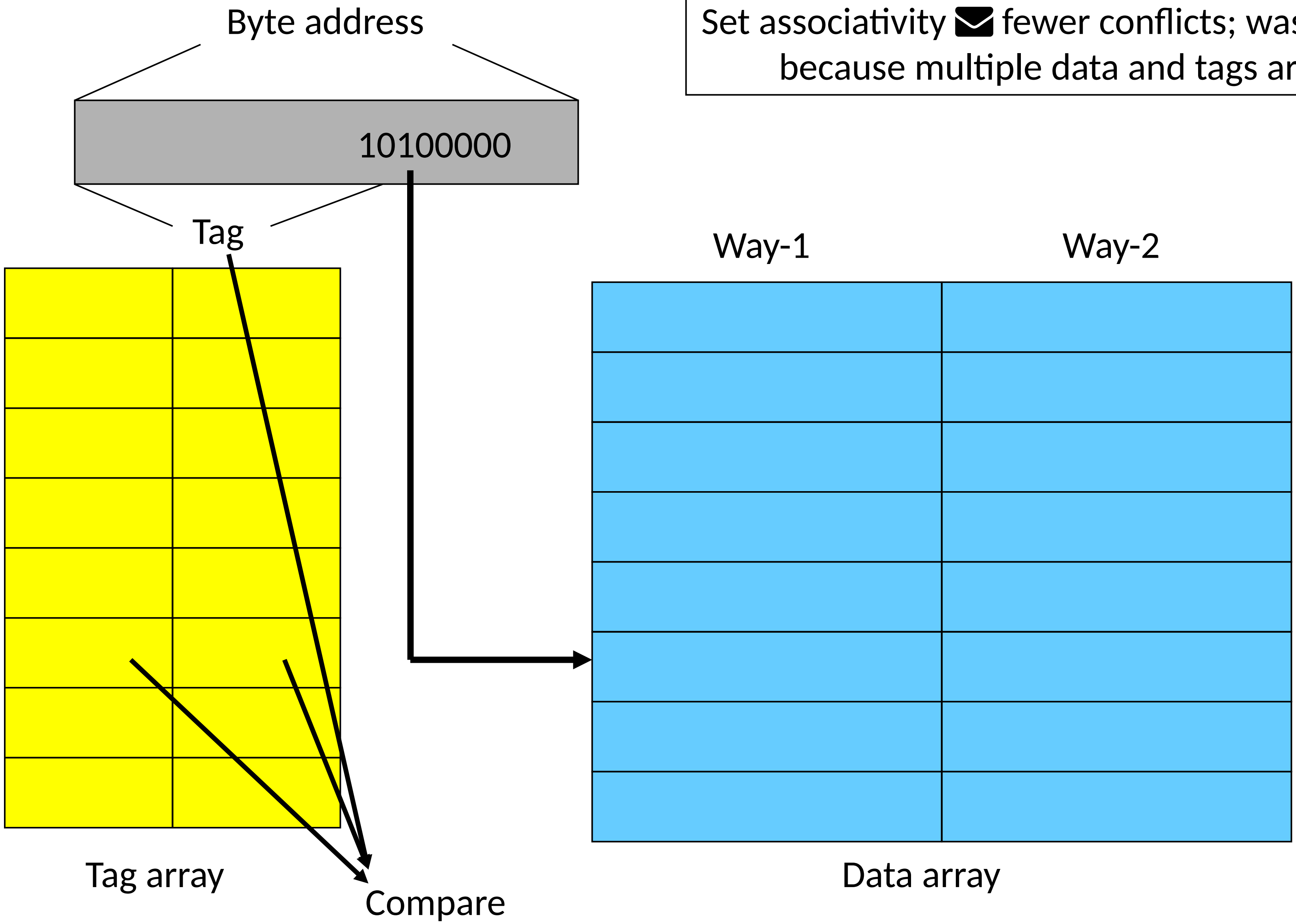
Increasing Line Size

A large cache line size ✉ smaller tag array,
fewer misses because of spatial locality



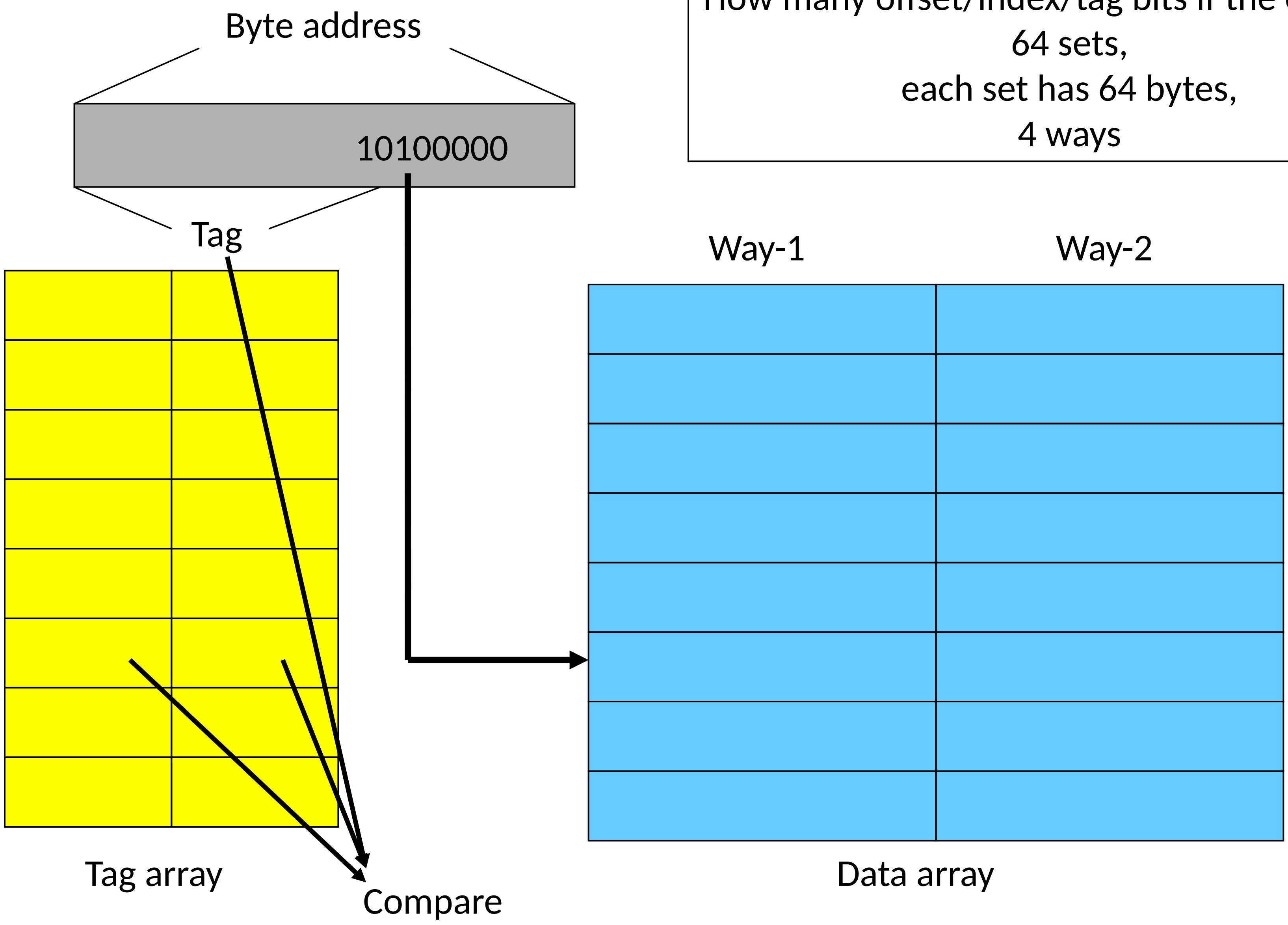
Associativity

Set associativity ✉ fewer conflicts; wasted power because multiple data and tags are read



Associativity

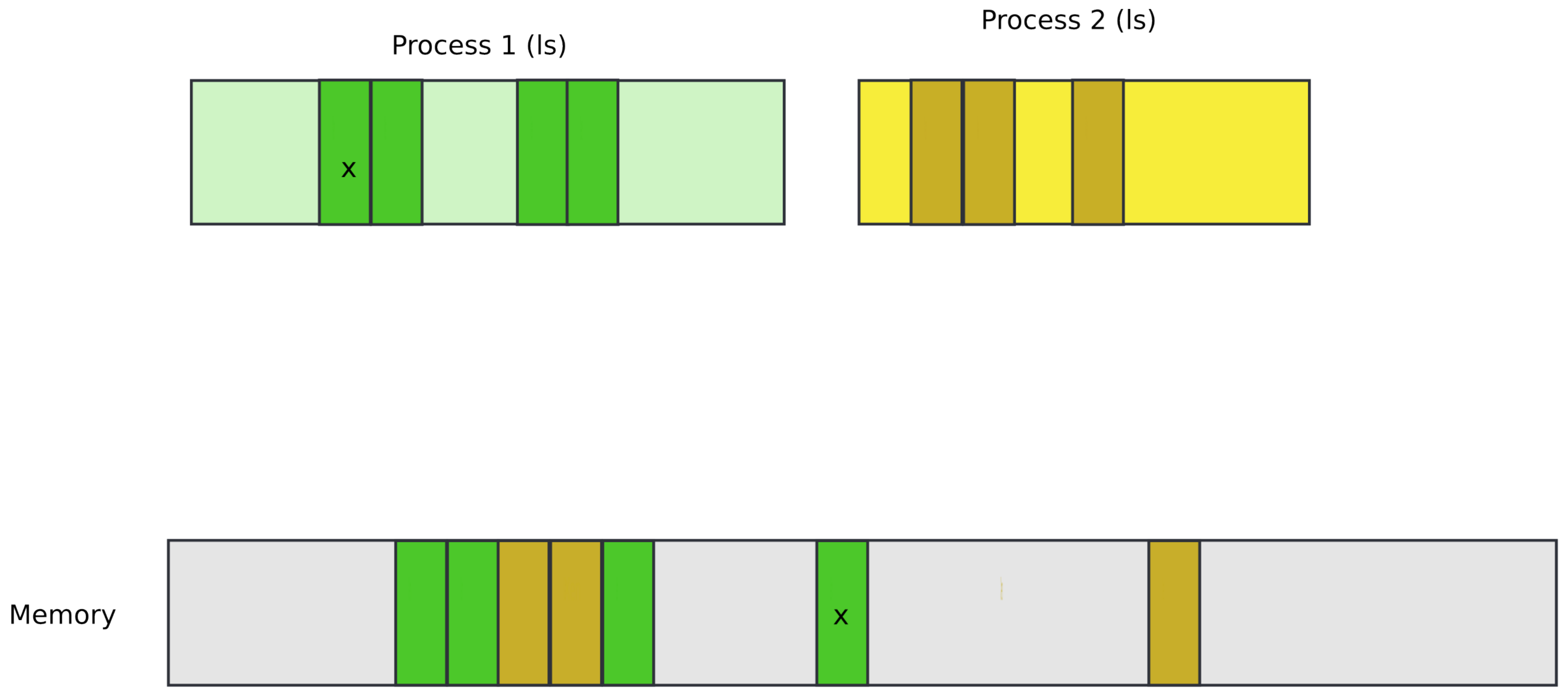
How many offset/index/tag bits if the cache has
64 sets,
each set has 64 bytes,
4 ways



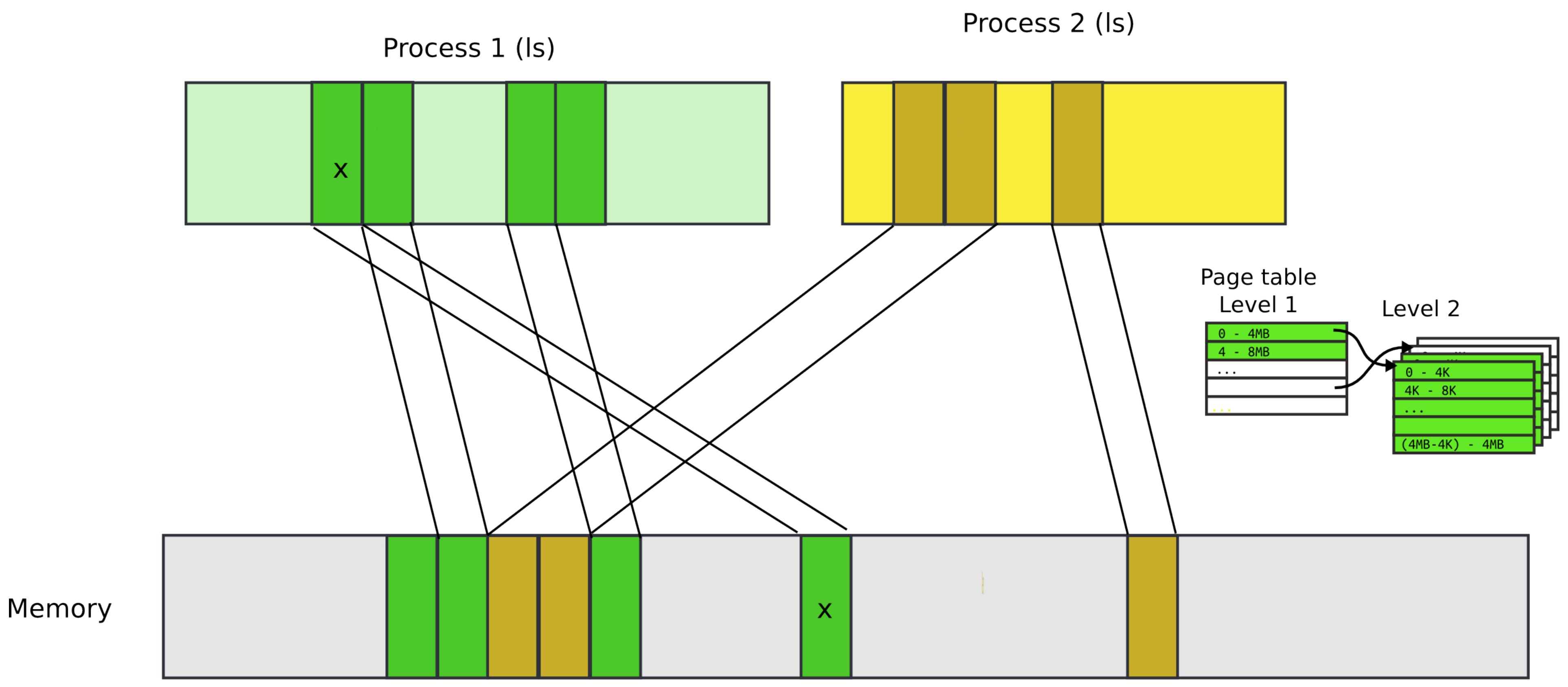
Memory and virtual memory

Paging

Pages



Pages


Process 1 (ls)
Process 2 (ls)
x
Page table
Level 1
Level 2
0 - 4MB
4 - 8MB
...
...
0 - 4K
4K - 8K
...
(4MB-4K) - 4MB
Memory
x

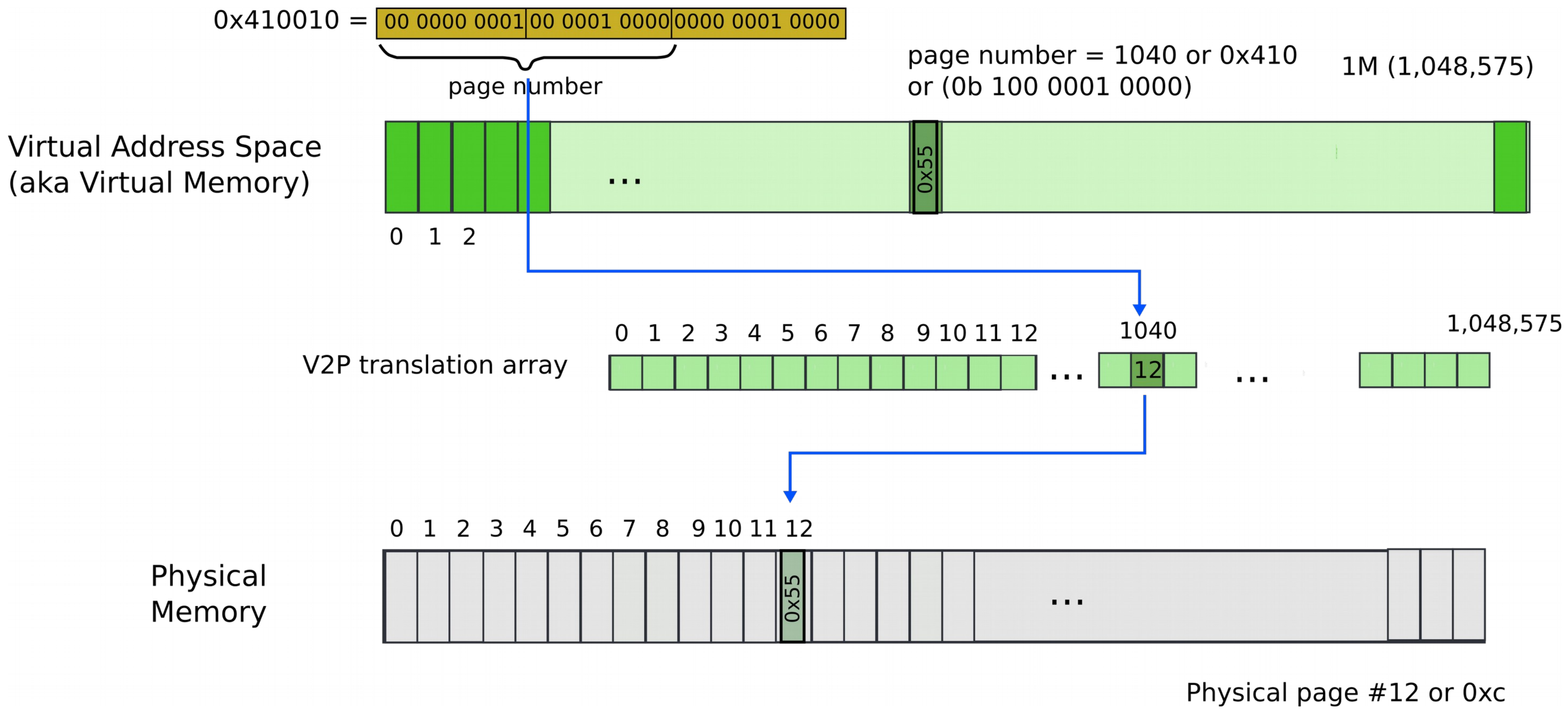
Paging idea

- Break up memory into 4096-byte chunks called pages
  - Modern hardware supports 2MB, 4MB, and 1GB pages
- Independently control mapping for each page of linear address space

- Compare with segmentation (single base + limit)
  - many more degrees of freedom

How can we build this translation mechanism?

Paging: naive approach: translation array



- Virtual address 0x410010

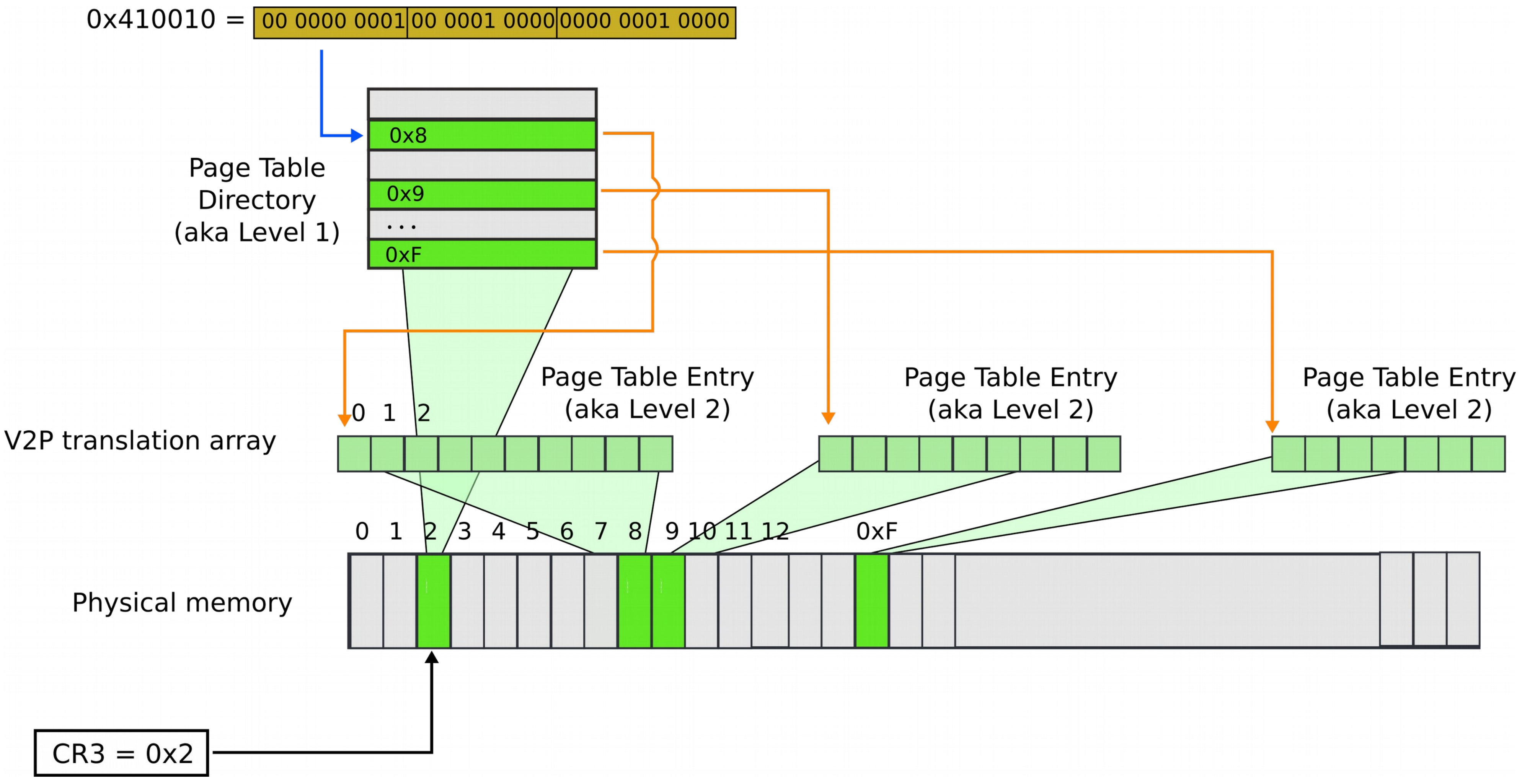
What is wrong?

What is wrong?

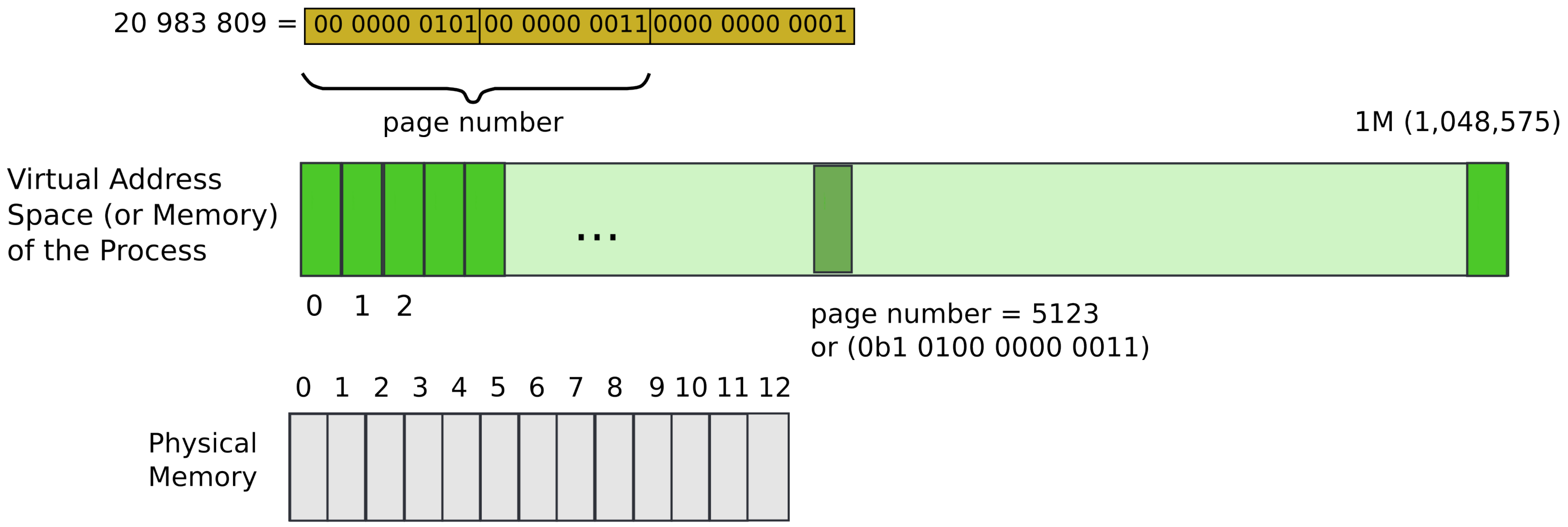
- We need 4 bytes to relocate each page
  - 20 bits for physical page number
  - 12 bits of access flags

  - Therefore, we need array of 4 bytes x 1M entries
    - 4MBs

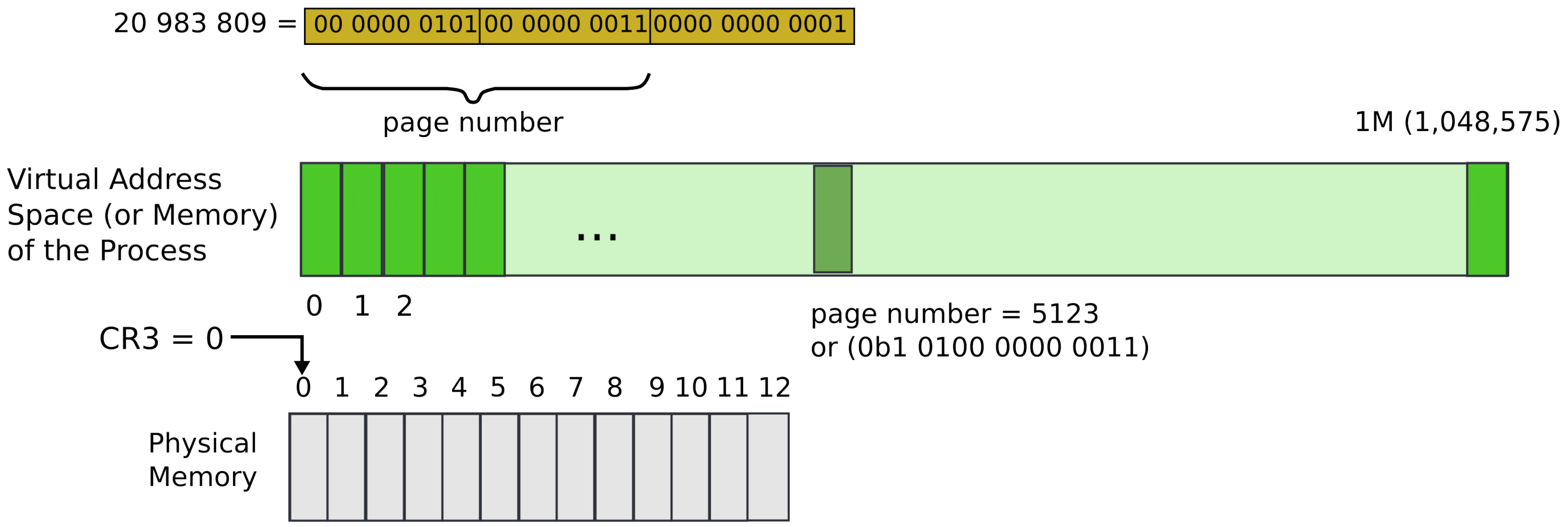
Paging: page table


0x410010 = 00 0000 0001 | 00 0001 0000 | 0000 0001 0000
Page Table Directory (aka Level 1)
0x8
0x9
. . .
0xF
Page Table Entry (aka Level 2)
Page Table Entry (aka Level 2)
Page Table Entry (aka Level 2)
V2P translation array
0 1 2
0 1 2 3 4 5 6 7 8 9 10 11 12 0xF
Physical memory
CR3 = 0x2

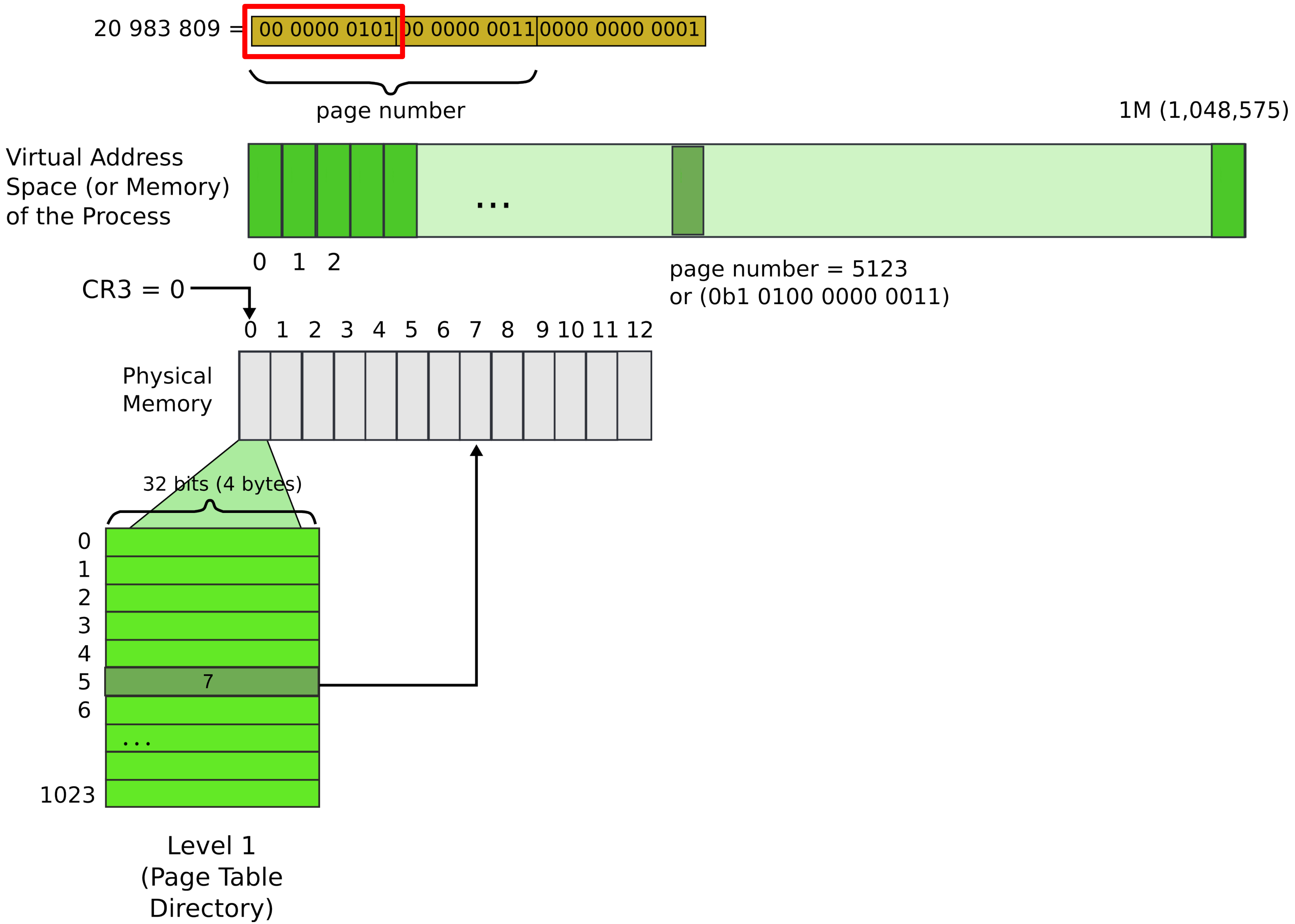
```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```



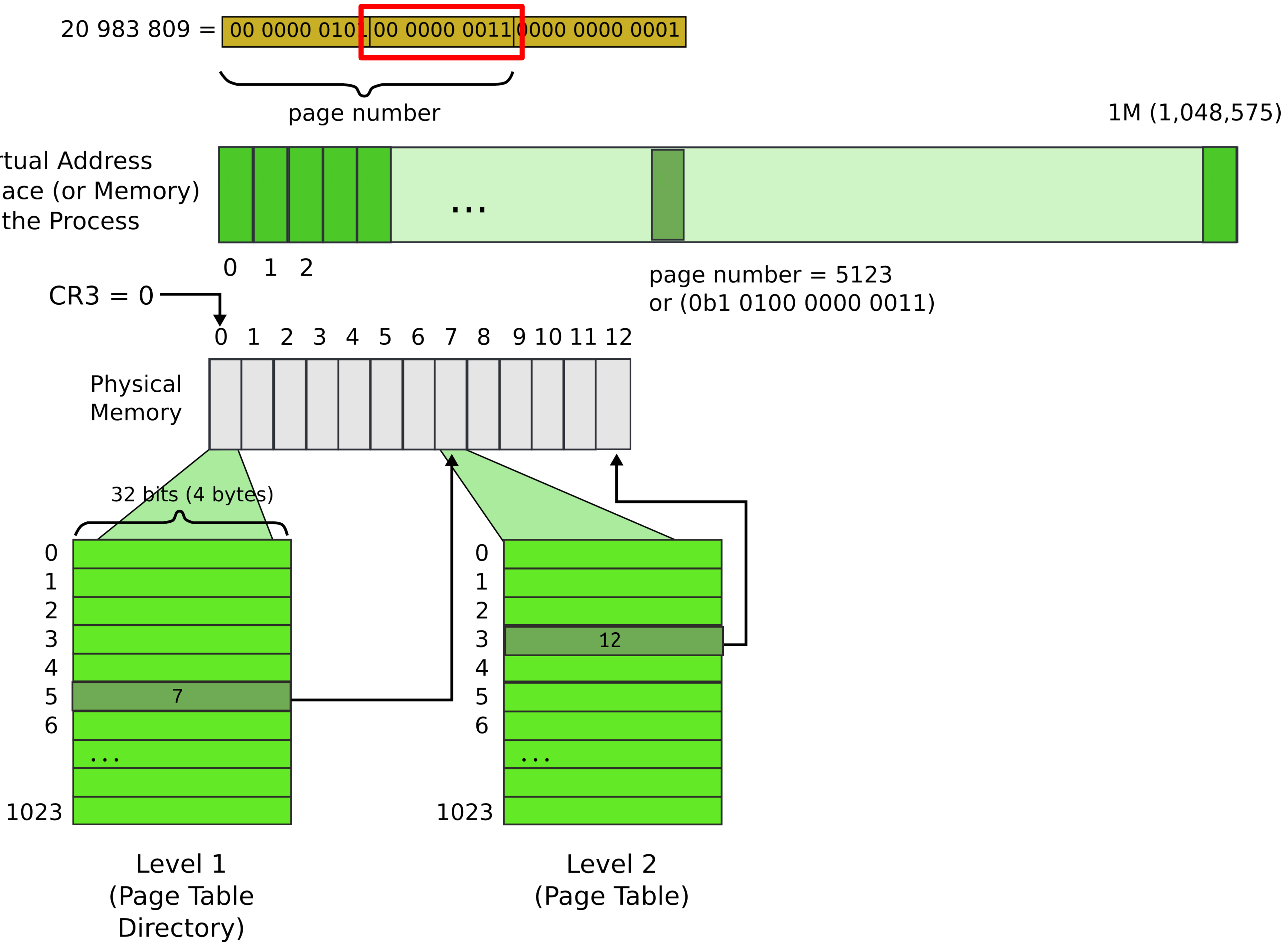
```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```



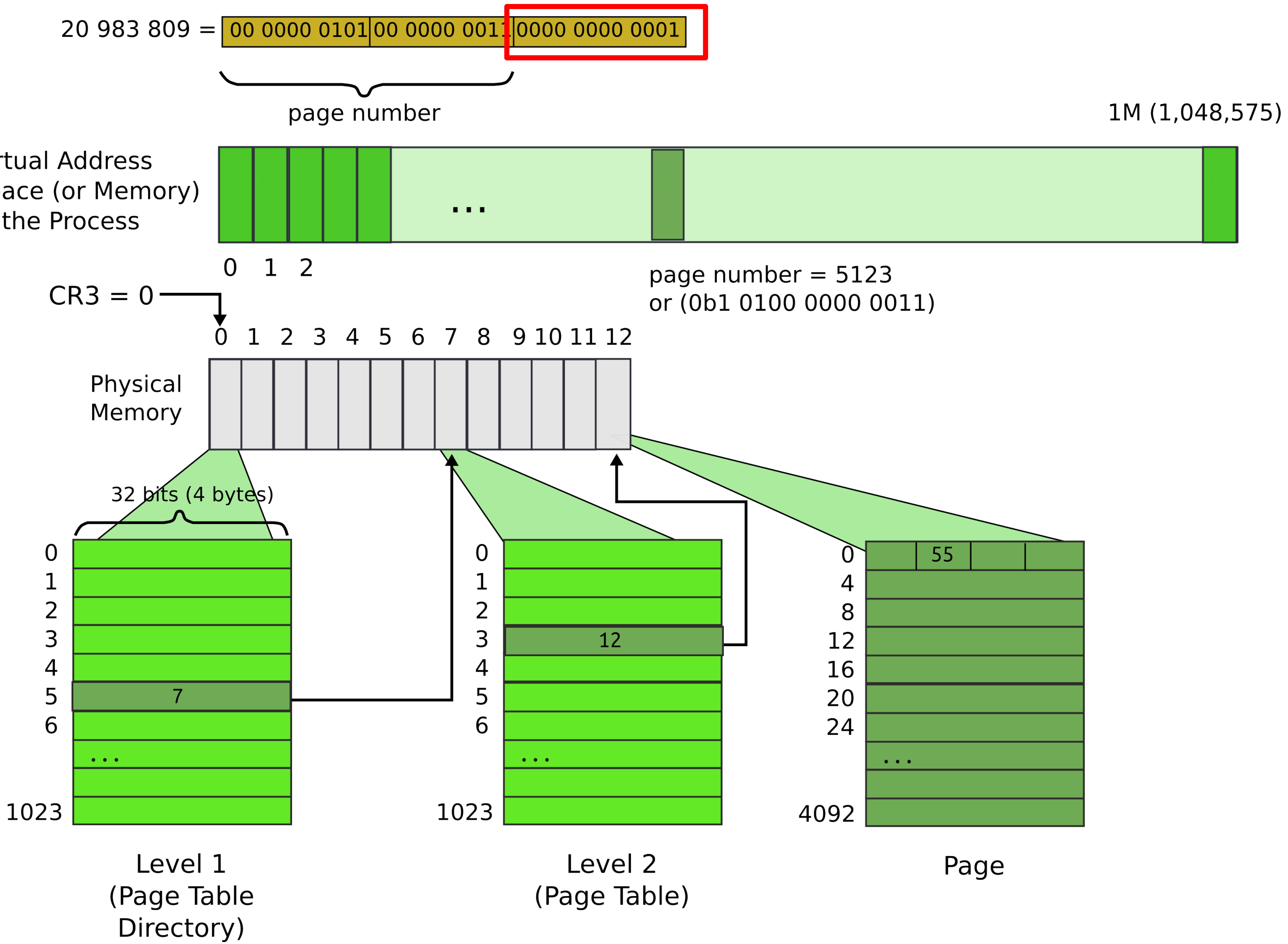
```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```



```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```

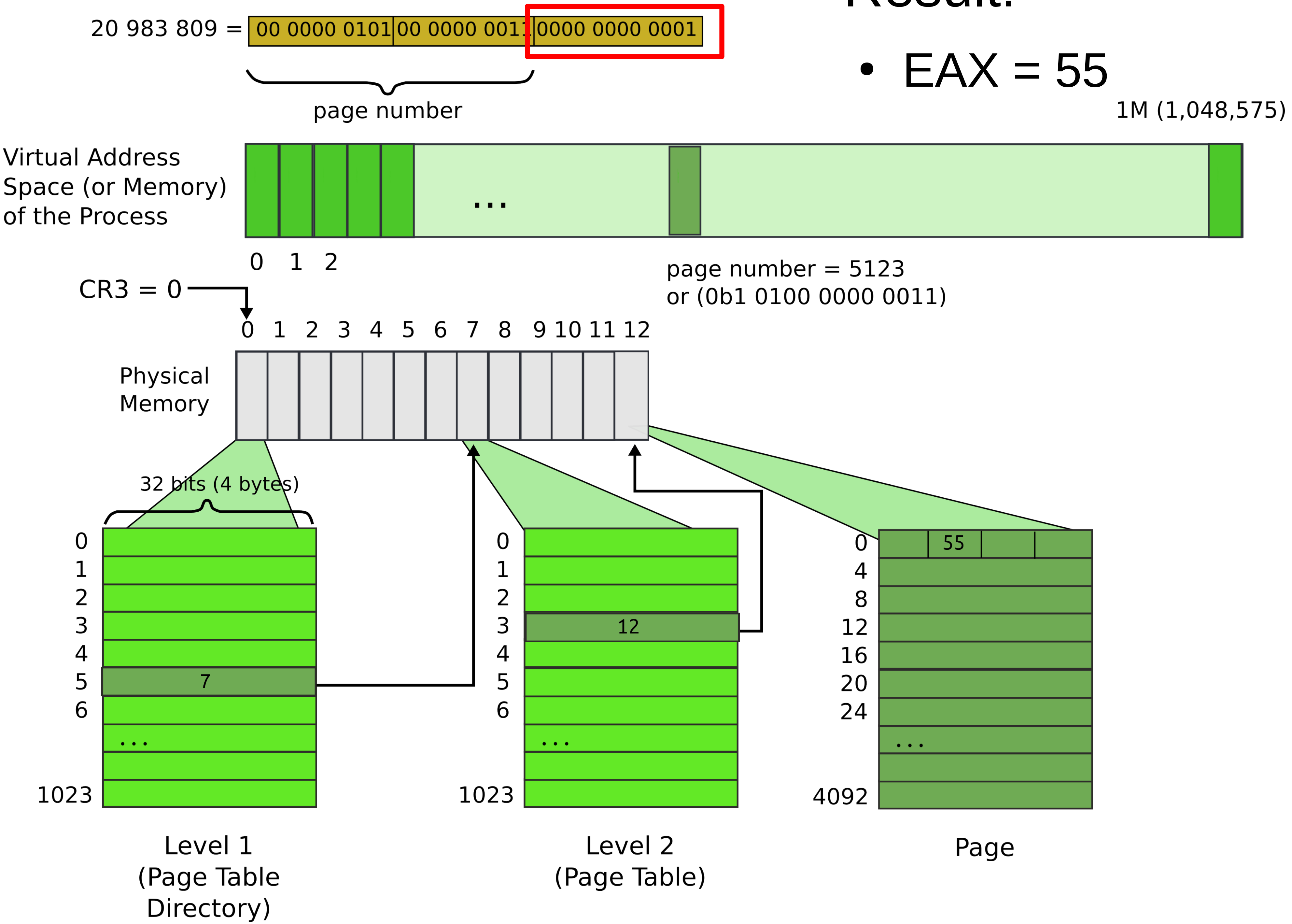


```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```



```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```

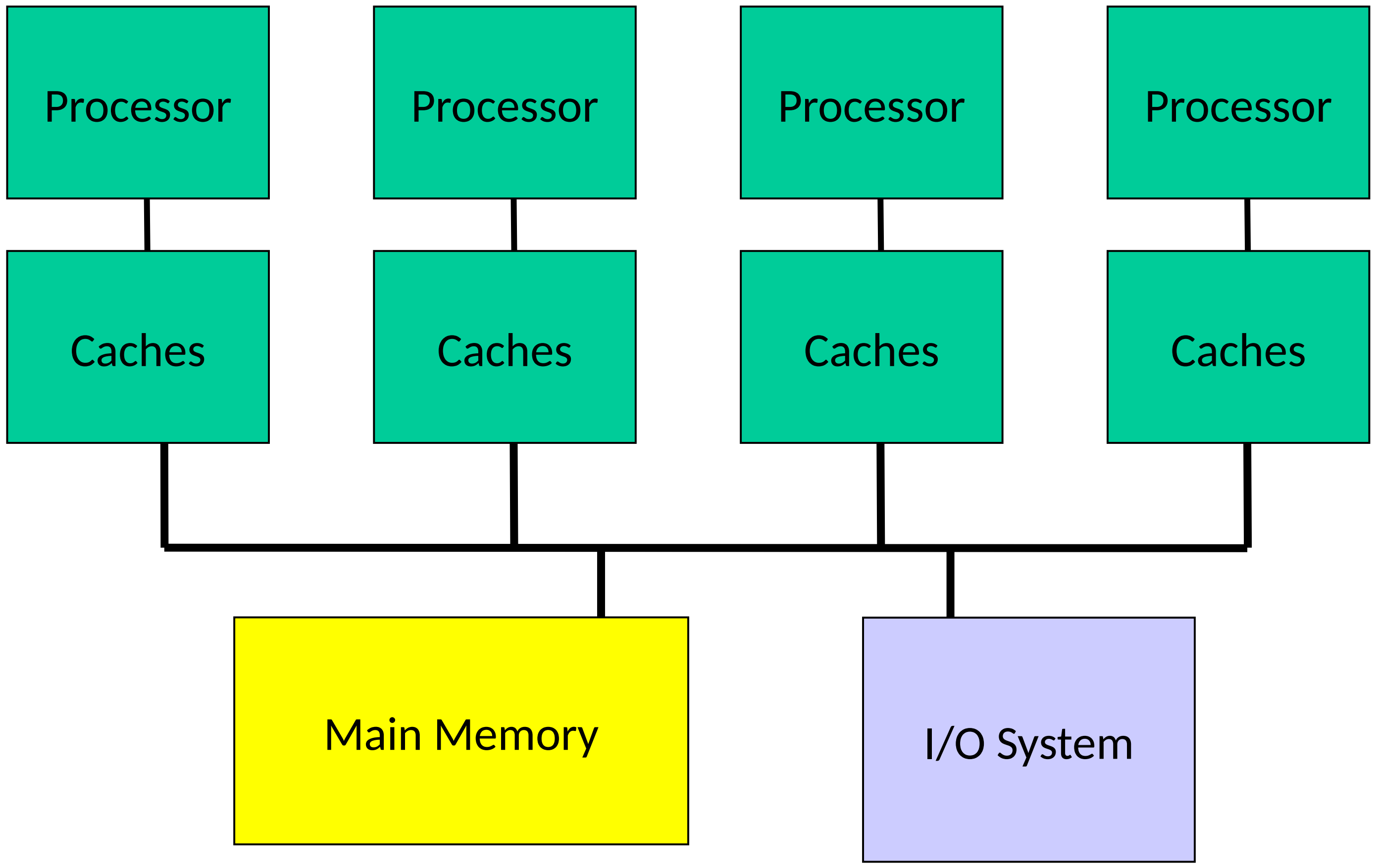
- Result:
  - EAX = 55



Cache-coherence

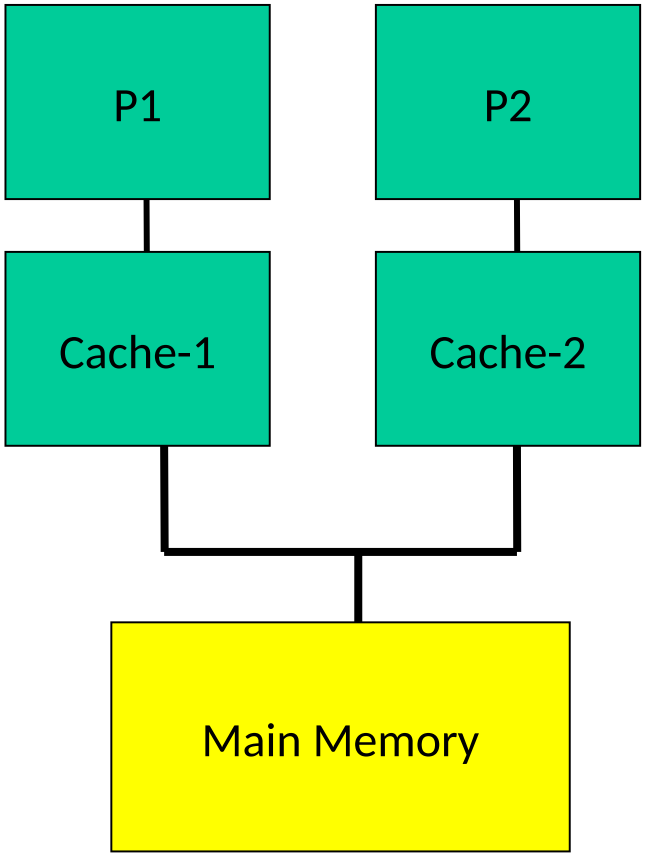
Snooping-Based Protocols

- Three states for a block: invalid, shared, modified
- A write is placed on the bus and sharers invalidate themselves
- The protocols are referred to as MSI, MESI, etc.



Example

- P1 reads X: not found in cache-1, request sent on bus, memory responds, X is placed in cache-1 in shared state
- P2 reads X: not found in cache-2, request sent on bus, everyone snoops this request, cache-1does nothing because this is just a read request, memory responds, X is placed in cache-2 in shared state



- P1 writes X: cache-1 has data in shared state (shared only provides read perms), request sent on bus, cache-2 snoops and then invalidates its copy of X, cache-1 moves its state to modified
- P2 reads X: cache-2 has data in invalid state, request sent on bus, cache-1 snoops and realizes it has the only valid copy, so it downgrades itself to shared state and responds with data, X is placed in cache-2 in shared state, memory is also updated

Example

Request	Cache Hit/Miss	Request on the bus	Who responds	State in Cache 1	State in Cache 2	State in Cache 3	State in Cache 4
				Inv	Inv	Inv	Inv
P1: Rd X	Miss	Rd X	Memory	S	Inv	Inv	Inv
P2: Rd X	Miss	Rd X	Memory	S	S	Inv	Inv
P2: Wr X	Perms Miss	Upgrade X	No response. Other caches invalidate.	Inv	M	Inv	Inv
P3: Wr X	Write Miss	Wr X	P2 responds	Inv	Inv	M	Inv
P3: Rd X	Read Hit	-	-	Inv	Inv	M	Inv
P4: Rd X	Read Miss	Rd X	P3 responds. Mem wrtbk	Inv	Inv	S	S

Before midterm

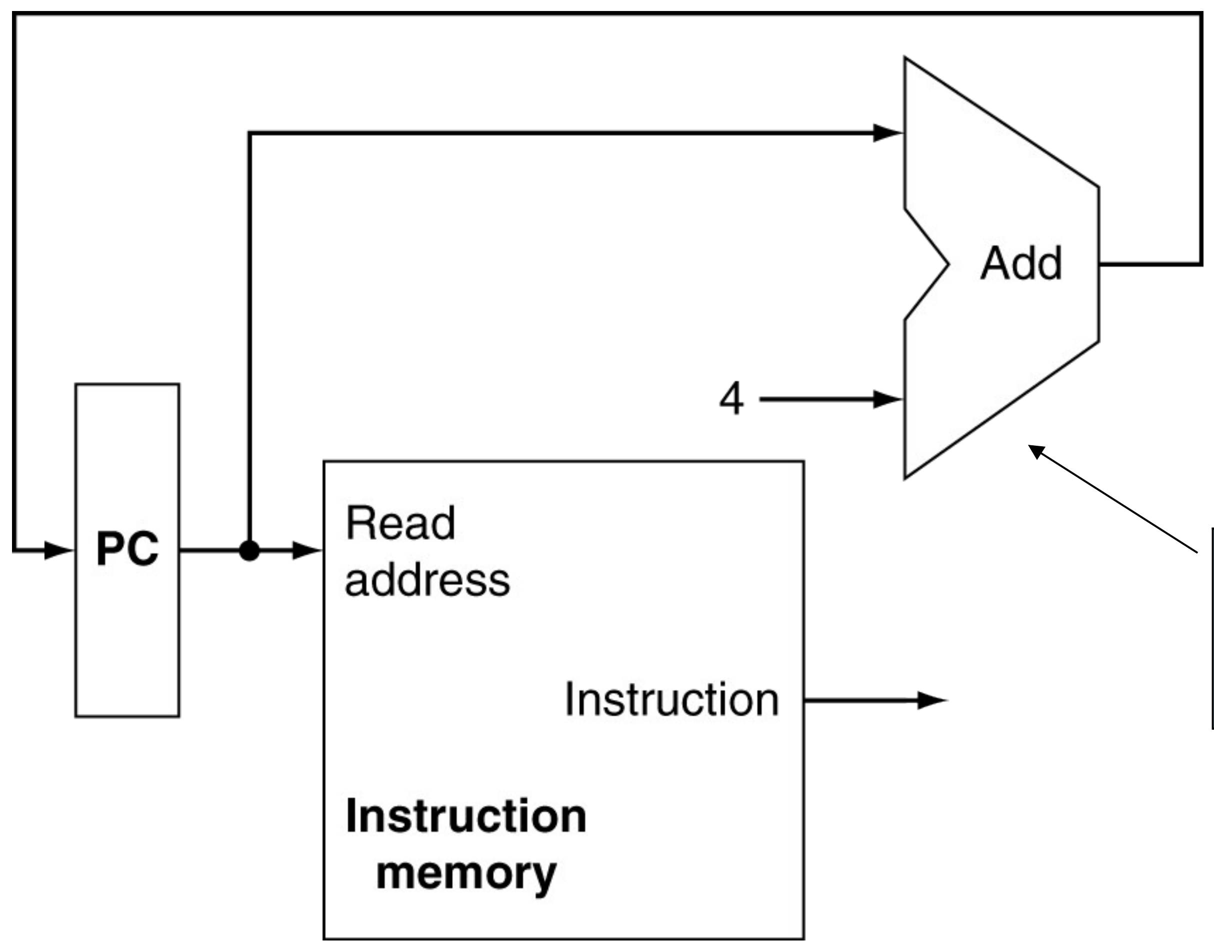
Simple program: String Copy

- MIPS code:

```
strcpy:
     addi $sp, $sp, -4       # adjust stack for 1 item
     sw   $s0, 0($sp)        # save $s0
     add  $s0, $zero, $zero # i = 0
L1:  add  $t1, $s0, $a1      # addr of y[i] in $t1
     lbu  $t2, 0($t1)        # $t2 = y[i]
     add  $t3, $s0, $a0      # addr of x[i] in $t3
     sb   $t2, 0($t3)        # x[i] = y[i]
     beq  $t2, $zero, L2     # exit loop if y[i] == 0
     addi $s0, $s0, 1        # i = i + 1
     j    L1                 # next iteration of loop
L2:  lw   $s0, 0($sp)        # restore saved $s0
     addi $sp, $sp, 4        # pop 1 item from stack
     jr   $ra                # and return
```

- Where is this program stored?
- How does it get executed?

Instruction Fetch



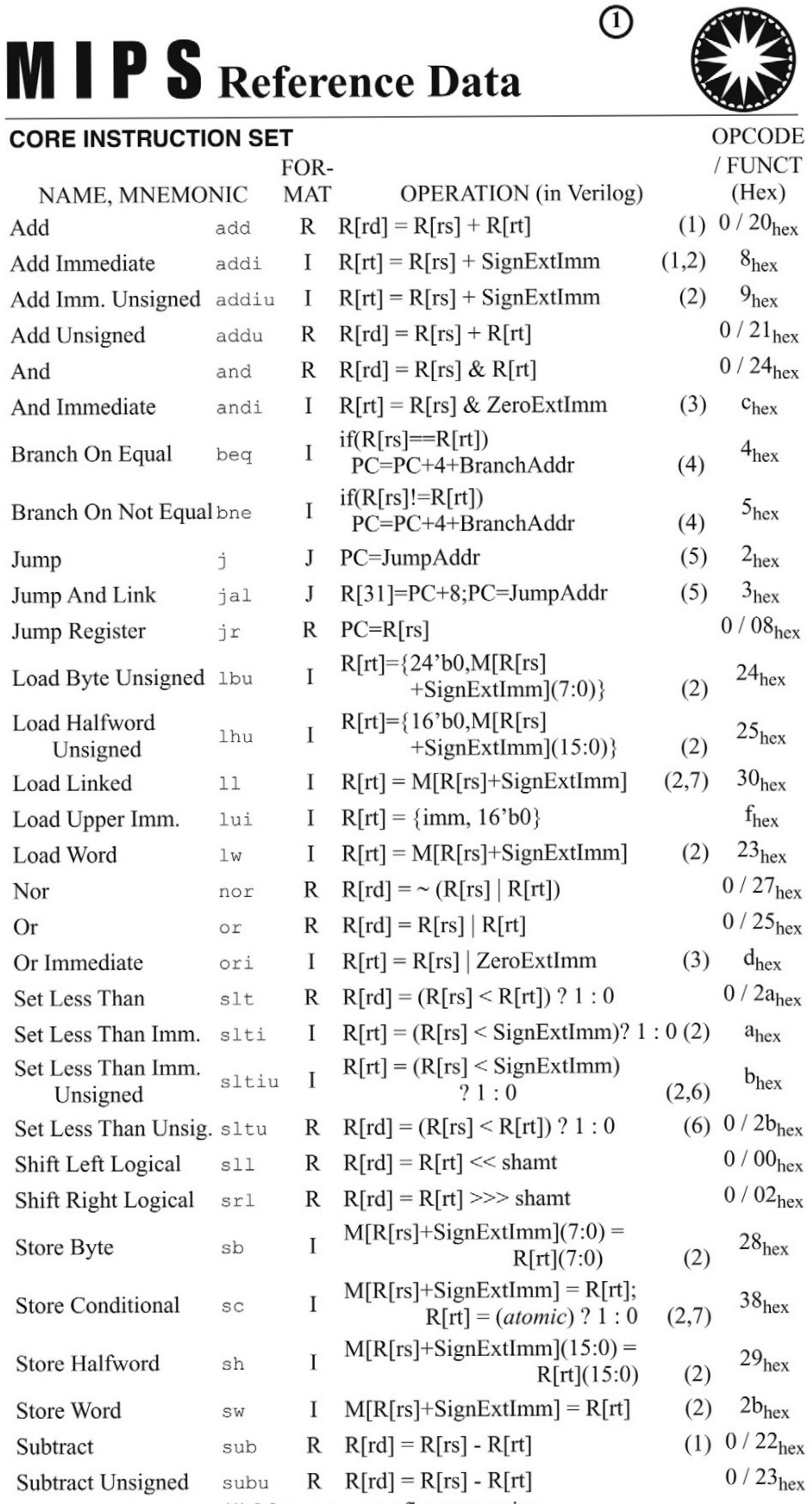
32-bit register

Increment by 4 for next instruction

Three types of instructions

- R-Type
  - Add, and, jr, nor, or, slt, sll, sub
- I-Type
  - addi, beq, bne, lw (and other loads), ori, slti, sw (and other stores)
- J-Type
  - j, jal

Fits on one page



①

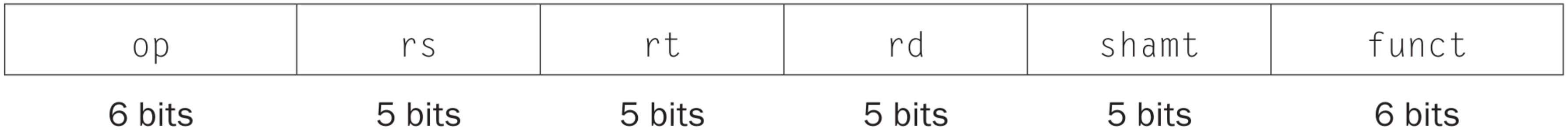
MIPS Reference Data

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	$0 / 20_{hex}$
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		$0 / 21_{hex}$
And	and	R	R[rd] = R[rs] & R[rt]		$0 / 24_{hex}$
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		$0 / 08_{hex}$
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		$0 / 27_{hex}$
Or	or	R	R[rd] = R[rs] | R[rt]		$0 / 25_{hex}$
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		$0 / 2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	$0 / 2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		$0 / 00_{hex}$
Shift Right Logical	srl	R	R[rd] = R[rt] >>> shamt		$0 / 02_{hex}$
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	$0 / 22_{hex}$
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		$0 / 23_{hex}$

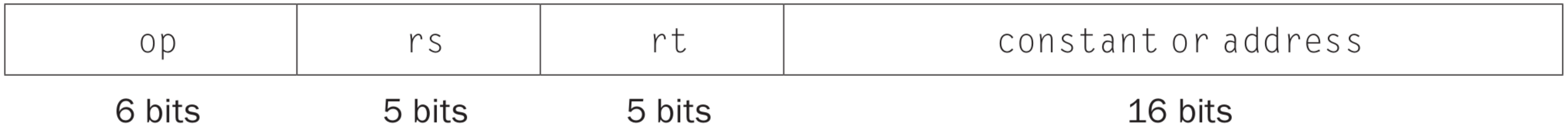
Three types of instructions

- R-Type



op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- I-Type

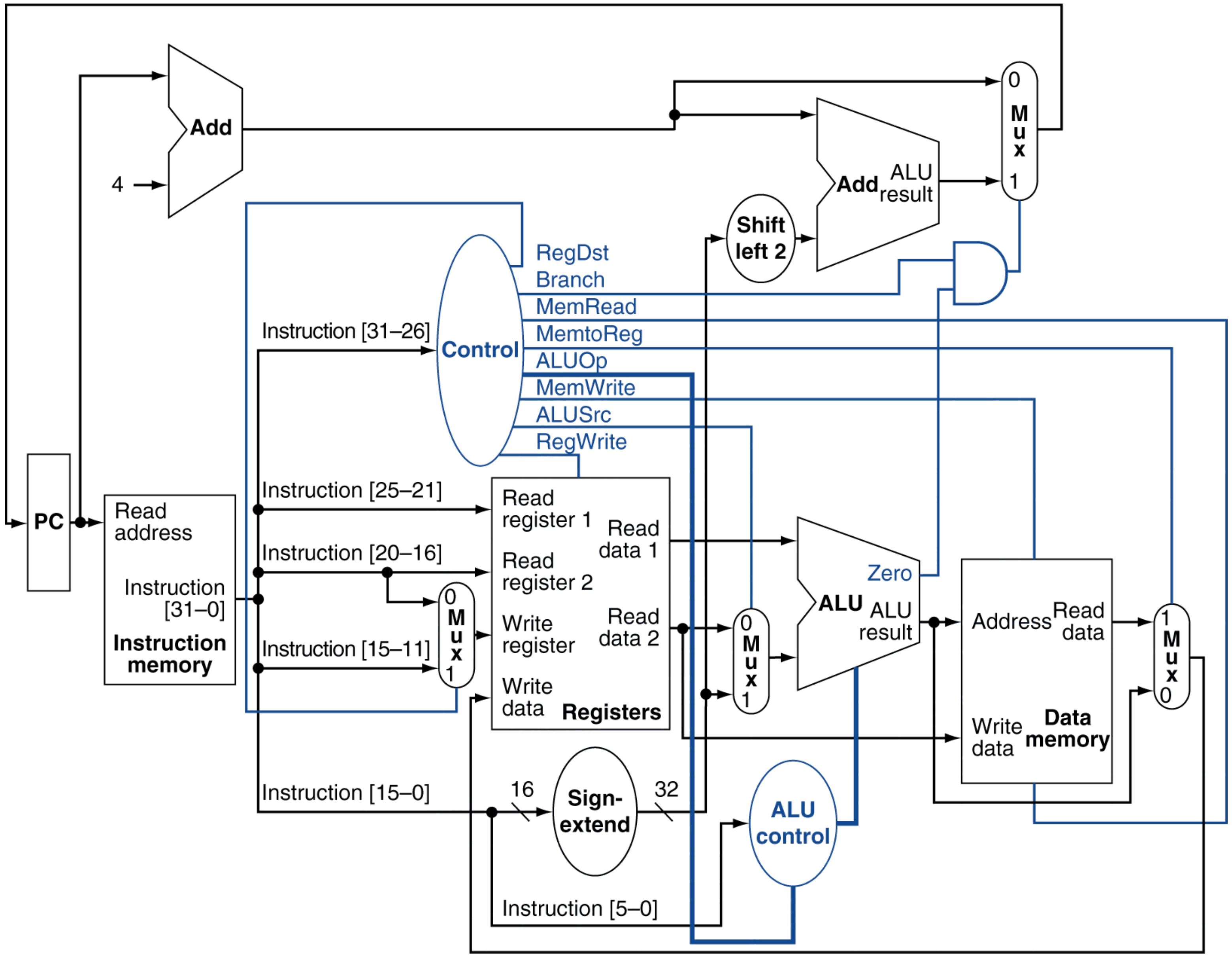


op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

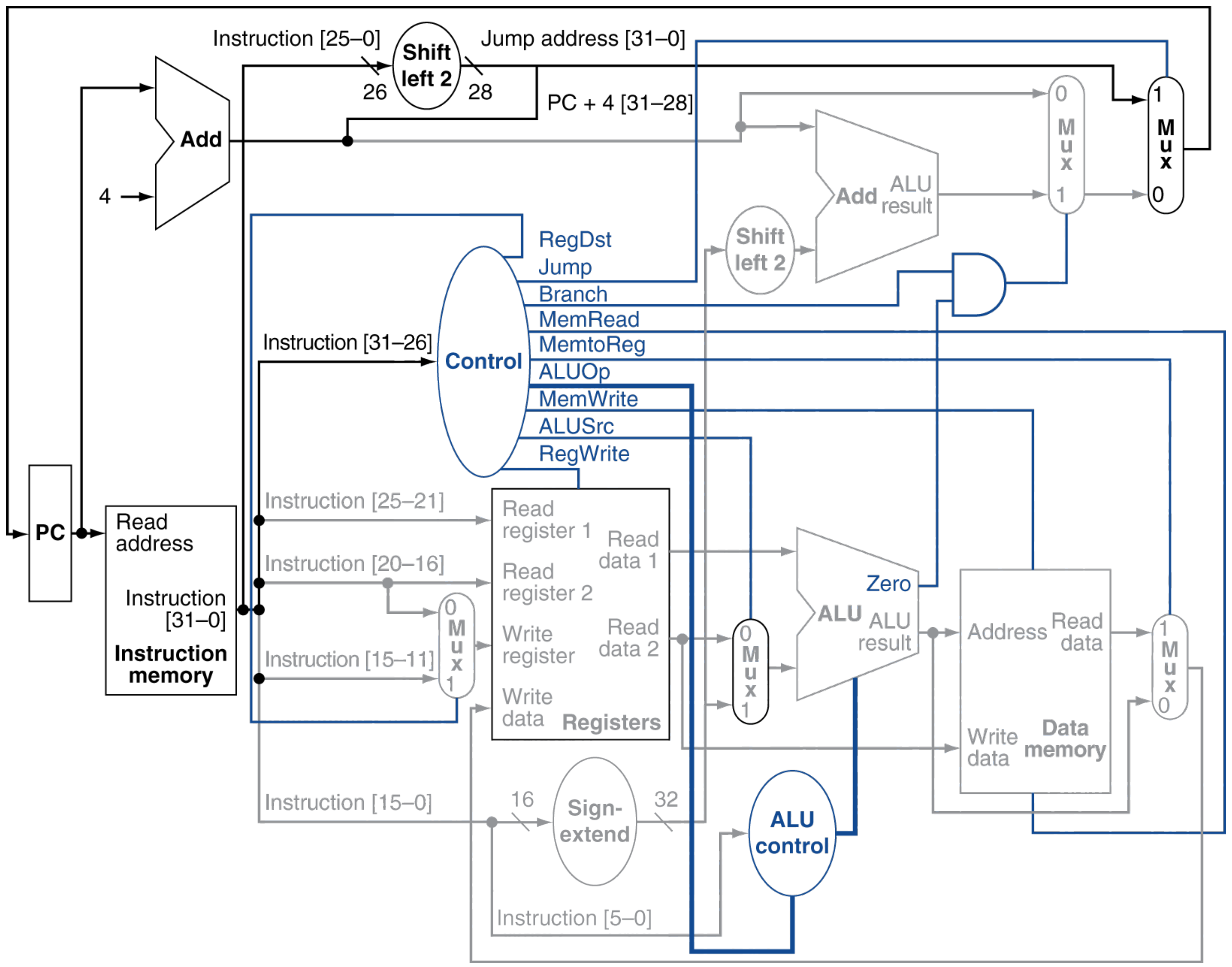
- J-Type

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

Datapath With Control



Datapath With Jumps Added



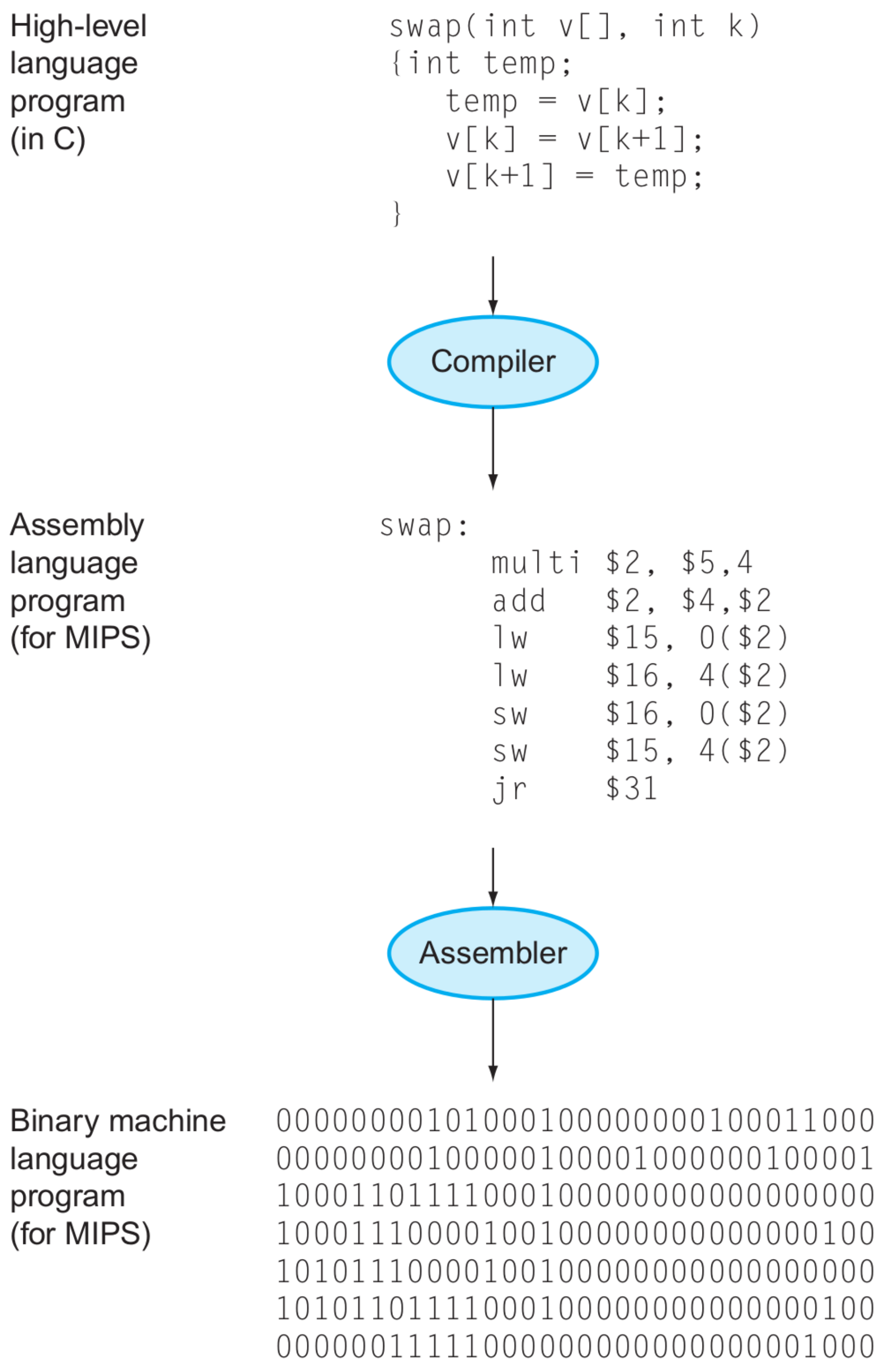
How do we get from high-level languages to MIPS instructions?

The HW/SW Interface

Application software

Systems software
(OS, compiler)

Hardware

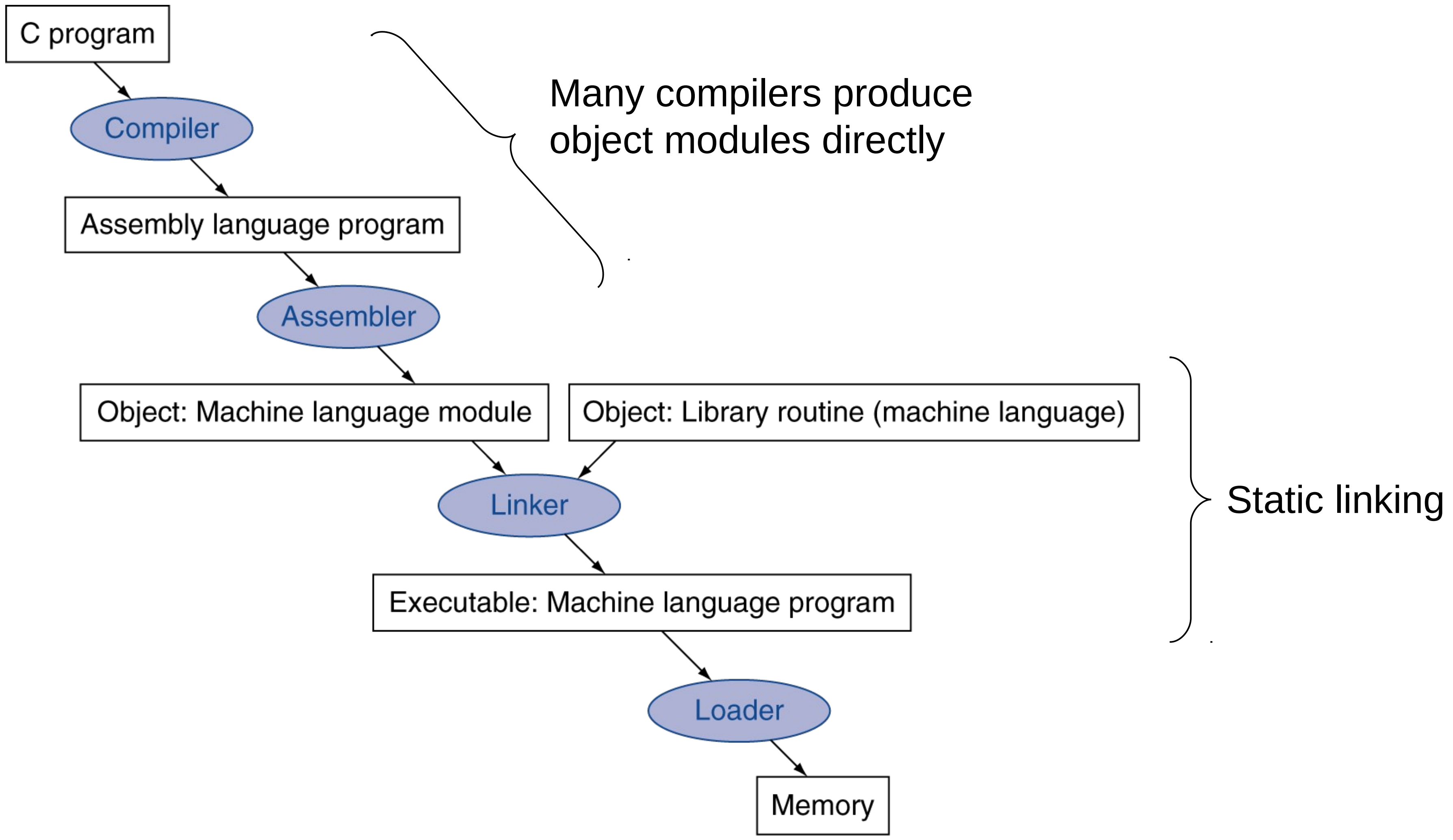


```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
      multi $2, $5,4
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

Translation and Startup


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

If then .. else …

```
if (i == j) f = g + h; else f = g - h;
```

- Assume that f,g, h, i, and j are in $s0, $s1, etc.

If then .. else …

```
if (i == j) f = g + h; else f = g - h;
```

- Assume that f,g, h, i, and j are in $s0, $s1, etc.

```
     bne $s3,$s4,Else # go to Else if i ≠ j

     add $s0,$s1,$s2  # f = g + h (skipped if i ≠ j)

     j Exit           # unconditional jump to Exit

Else:

     sub $s0,$s1,$s2  # f = g - h (skipped if i = j)

Exit:
```

Loops

```
while (save[i] == k)

    i += 1;
```

- Assume that i and k are in $s3 and $s5

Loops

```
while (save[i] == k)
    i += 1;
```

- Assume that i and k are in $s3 and $s5

```
Loop: sll $t1,$s3,2
      add $t1,$t1,$s6   # $t1 = address of save[i]
      lw $t0,0($t1)     # Temp reg $t0 = save[i]
      bne $t0,$s5, Exit # go to Exit if save[i] ≠ k
      addi $s3,$s3,1    # i = i + 1
      j Loop            # go to Loop
Exit:
```

Procedures

Calling functions

```
// some code...
foo();
// more code..
```

- $ra contains information for how to return from a subroutine
    - i.e., from foo()

- Functions can be called from different places in the program

```
if (a == 0) {
    foo();
    …
} else {
    foo();
    …
}
```

Procedure Call Instructions

- Procedure call: jump and link

```
jal ProcedureLabel
```

  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: jump register

```
jr $ra
```

  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Calling conventions

- Goal: re-entrant programs
  - How to pass arguments
    - On the stack?
    - In registers?
  - How to return values
    - On the stack?
    - In registers?
  - What registers have to be preserved
    - All? Some subset?
- Conventions differ from compiler, optimizations, etc.

Passing arguments

- First 4 arguments in registers
  - $a0 - $a3
- Other arguments on the stack
- Return values in registers
  - $v0 - $v1

Preserving registers

- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee

Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

Leaf Procedure Example

- MIPS code:

```
leaf_example:
    addi $sp, $sp, -4       Save $s0 on stack
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3      Procedure body
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero    Result
    lw   $s0, 0($sp)        Restore $s0
    addi $sp, $sp, 4
    jr   $ra                Return
```

Recursive invocations

```
foo(int a) {
    if (a == 0)
        return;
    a--;
    foo(a);
    return;
}

foo(4);
```

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
  if (n < 1) return f;
  else return n * fact(n - 1);
}
```

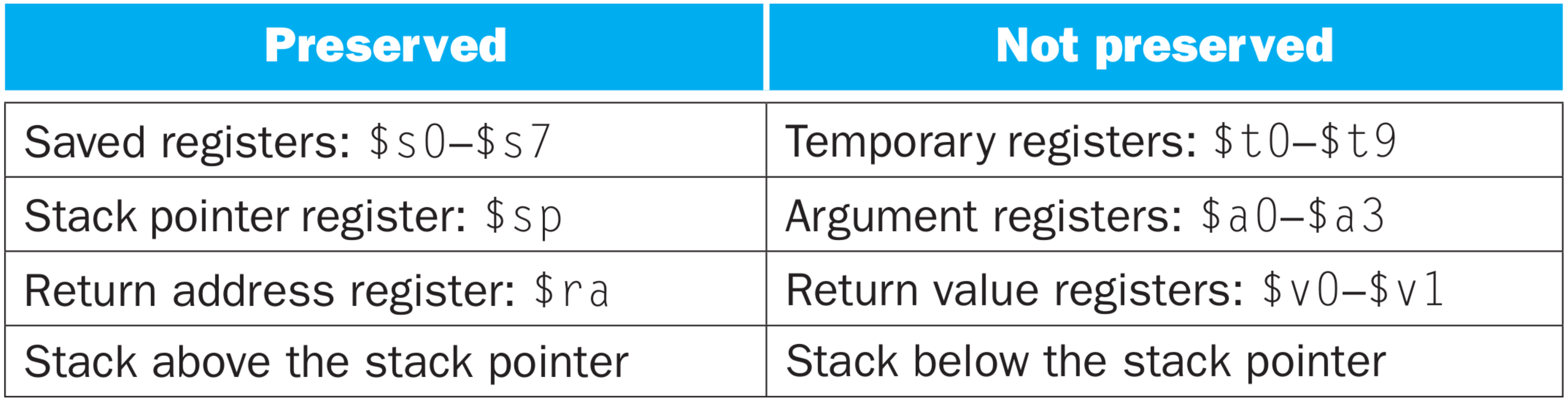
  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

- MIPS code:

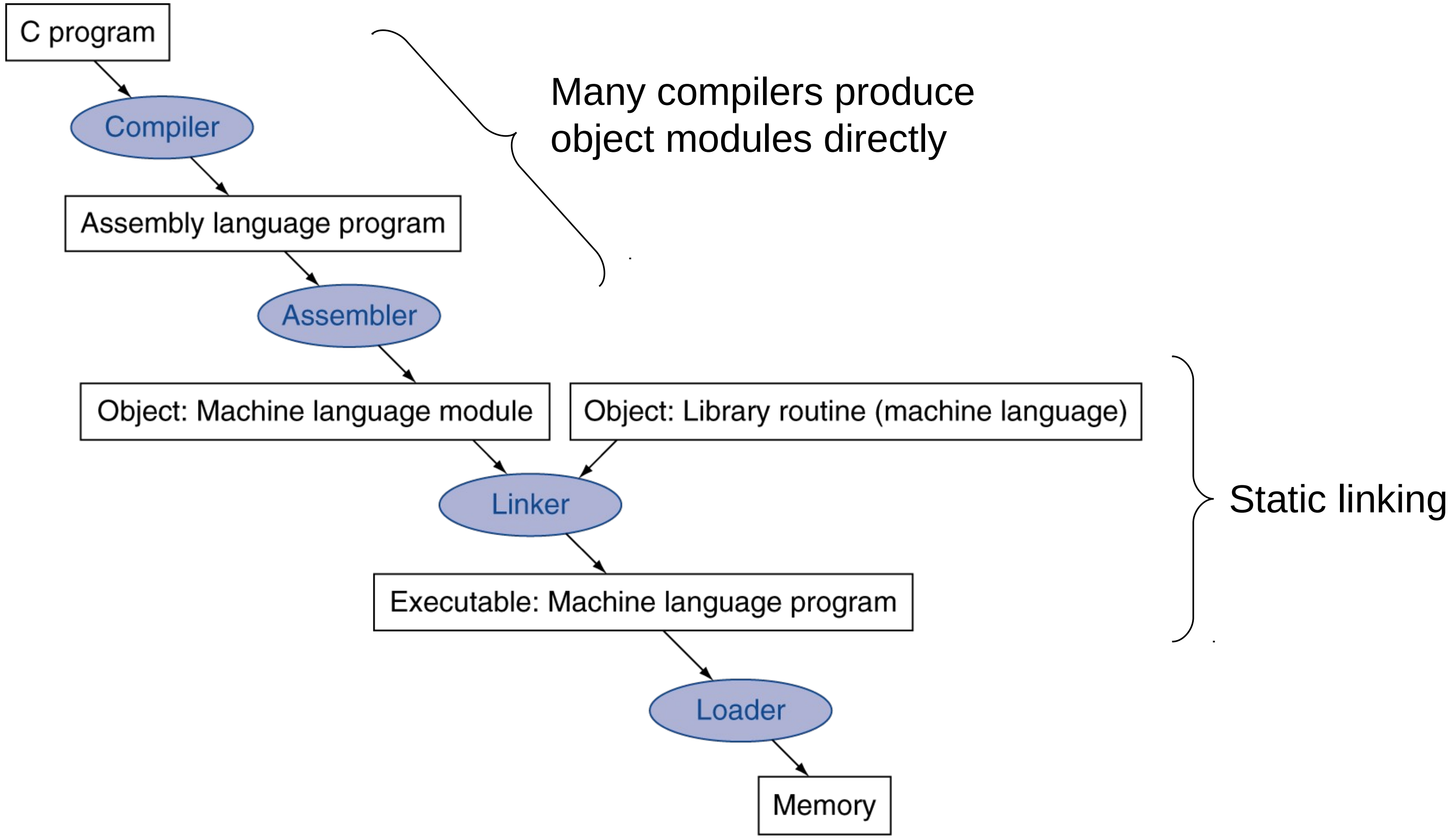
```
fact:
    addi $sp, $sp, -8     # adjust stack for 2 items
    sw   $ra, 4($sp)      # save return address
    sw   $a0, 0($sp)      # save argument
    slti $t0, $a0, 1      # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8      #   pop 2 items from stack
    jr   $ra              #   and return
L1: addi $a0, $a0, -1     # else decrement n
    jal  fact             # recursive call
    lw   $a0, 0($sp)      # restore original n
    lw   $ra, 4($sp)      #   and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra              # and return
```

Calling convention (again)

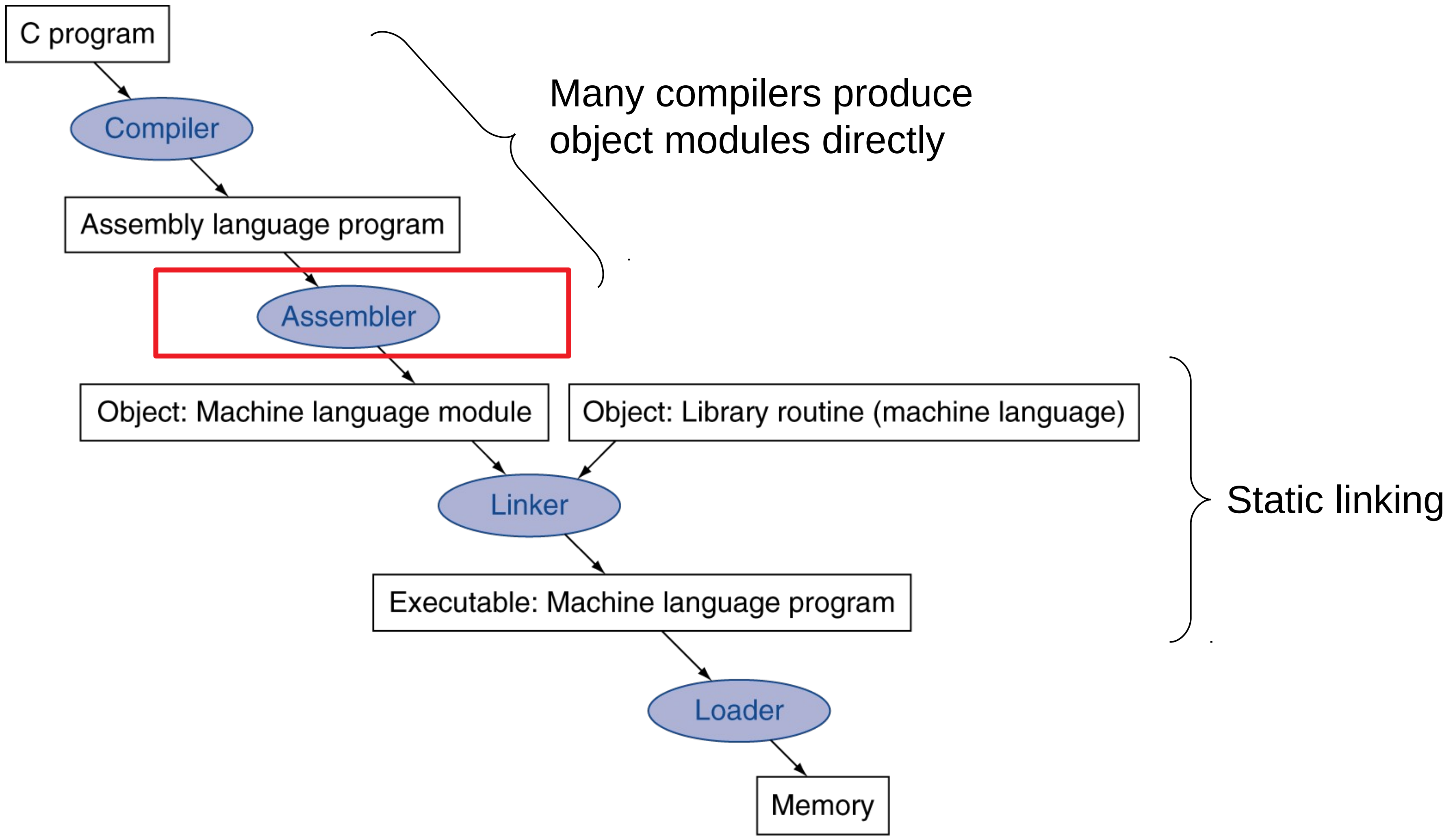


Preserved	Not preserved
Saved registers: `$s0–$s7`	Temporary registers: `$t0–$t9`
Stack pointer register: `$sp`	Argument registers: `$a0–$a3`
Return address register: `$ra`	Return value registers: `$v0–$v1`
Stack above the stack pointer	Stack below the stack pointer

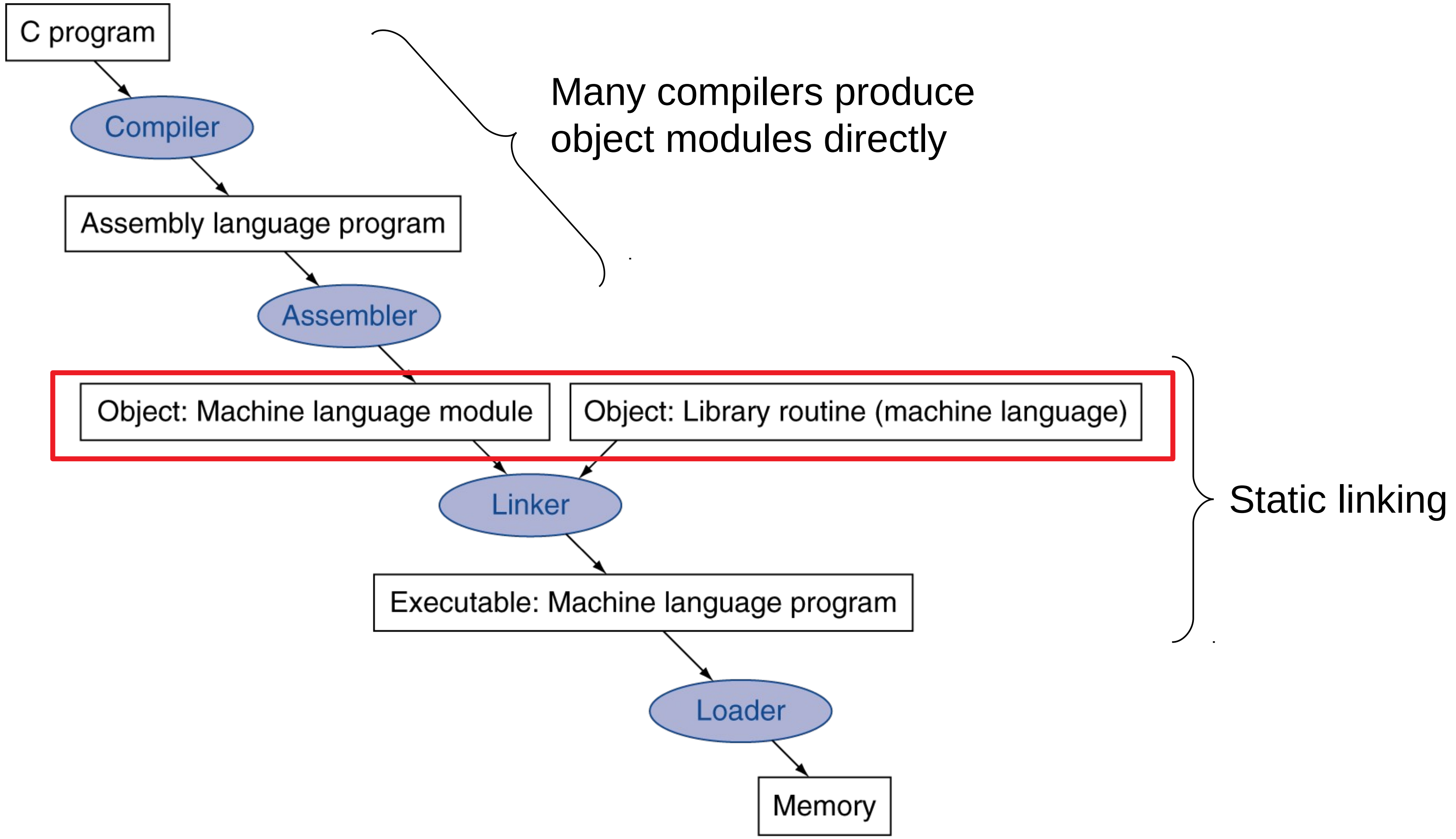
Translation and Startup


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

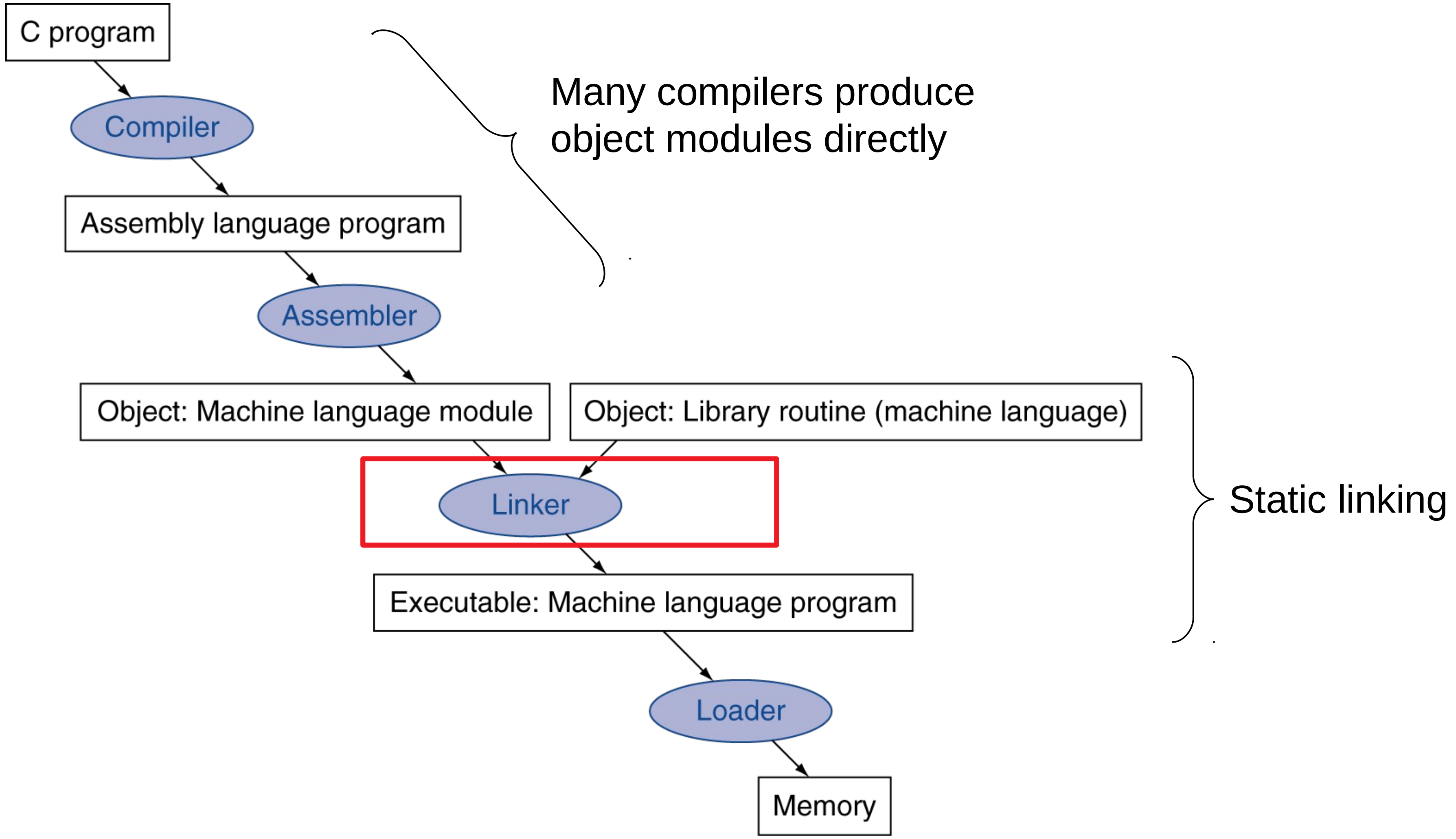
Translation and Startup

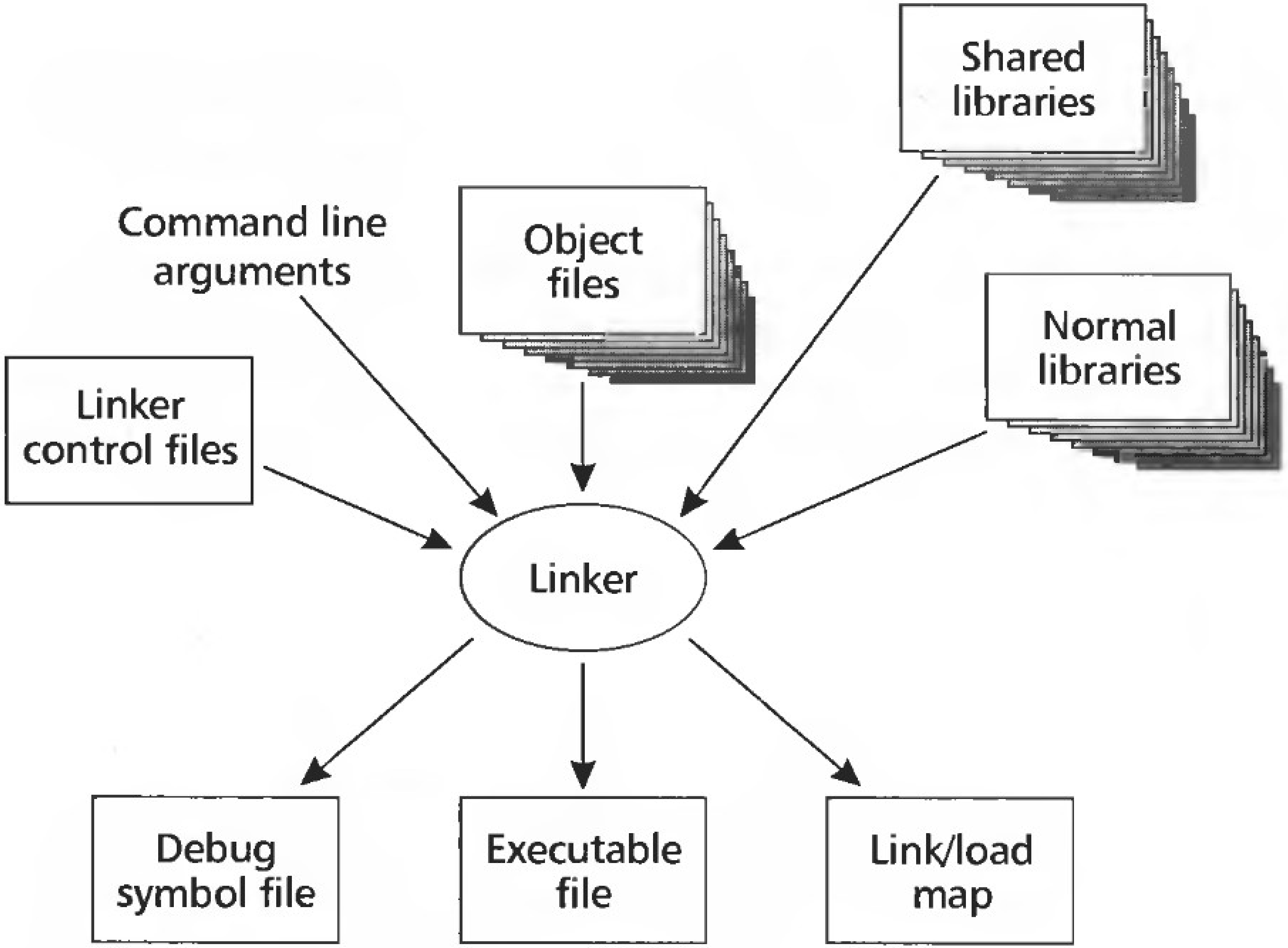

C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

Translation and Startup


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

Translation and Startup


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking


Shared libraries
Command line arguments
Object files
Normal libraries
Linker control files
Linker
Debug symbol file
Executable file
Link/load map

Relocation
(What needs to be done to move the program in memory?)

What types of variables do you know?

- Or where these variables are allocated in memory?

What types of variables do you know?

- Global variables
  - Initialized → data section
  - Uninitalized → BSS
- Dynamic variables
  - Heap
- Local variables
  - Stack

Global variables

```
#include <stdio.h>

char hello[] = "Hello";
int main(int ac, char **av)
{
        static char world[] = "world!";
        printf("%s %s\n", hello, world);
        return 0;
}
```

Global variables

```c
#include <stdio.h>

char hello[] = "Hello";
int main(int ac, char **av)
{
    static char world[] = "world!";
    printf("%s %s\n", hello, world);
    return 0;
}
```

- Allocated in the data section
  - It is split in initialized (non-zero), and non-initialized (zero)
  - As well as read/write, and read only data section

Global variables

Dynamic variables (heap)

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char hello[] = "Hello";
int main(int ac, char **av)
{
    char world[] = "world!";
    char *str = malloc(64);
    memcpy(str, "beautiful", 64);
    printf("%s %s %s\n", hello, str, world);
    return 0;
}
```

Dynamic variables (heap)

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char hello[] = "Hello";
int main(int ac, char **av)
{
    char world[] = "world!";
    char *str = malloc(64);
    memcpy(str, "beautiful", 64);
    printf("%s %s %s\n", hello, str, world);
    return 0;
}
```

- Allocated on the heap
  - Special area of memory provided by the OS from where malloc() can allocate memory

Local variables

- Local variables

```c
#include <stdio.h>

char hello[] = "Hello";
int main(int ac, char **av)
{
	//static char world[] = "world!";
	char world[] = "world!";
	printf("%s %s\n", hello, world);
	return 0;
}
```

```
# "Hello World" in MIPS assembly
# From: http://labs.cs.upt.ro/labs/so2/html/resources/nachos-doc/mipsf.html

# All program code is placed after the
# .text assembler directive
.text

# Declare main as a global function
.globl  main

# The label 'main' represents the starting point
main:
        # Run the print_string syscall which has code 4
        li      $v0,4           # Code for syscall: print_string
        la      $a0, msg        # Pointer to string (load the address of msg)
        syscall
        li      $v0,10          # Code for syscall: exit
        syscall

# All memory structures are placed after the
# .data assembler directive
        .data

        # The .asciiz assembler directive creates
        # an ASCII string in memory terminated by
        # the null character. Note that strings are
        # surrounded by double-quotes
msg:    .asciiz "Hello World!\n"
```

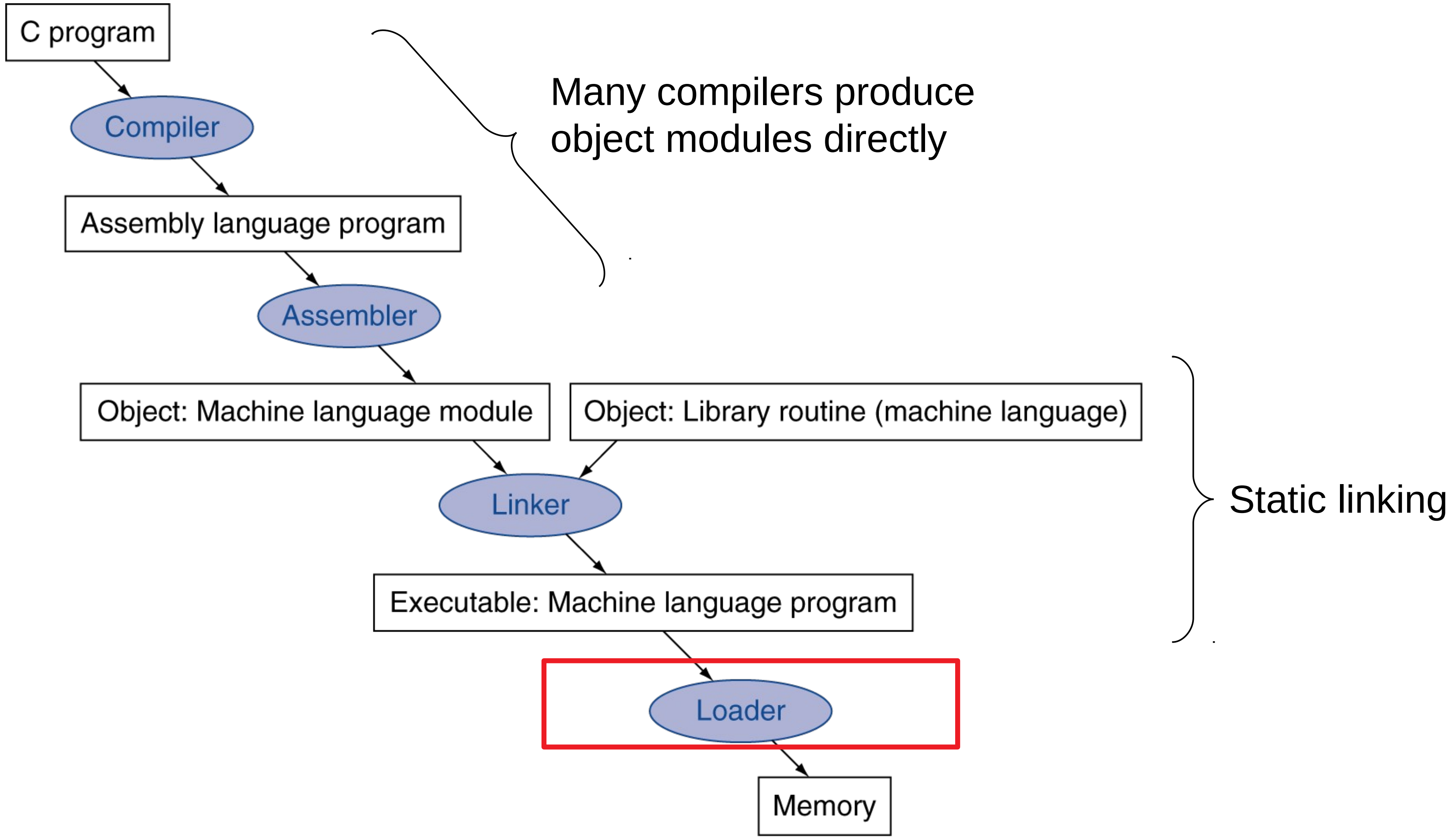
What needs to be relocated?

```
  User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)              ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4           ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4            ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2              ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2            ; 187: addu $a2 $a2 $v0
[00400014] 0c100009  jal 0x00400024 [main]      ; 188: jal main
[00400018] 00000000  nop                        ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10             ; 191: li $v0 10
[00400020] 0000000c  syscall                    ; 192: syscall # syscall 10 (exit)
[00400024] 34020004  ori $2, $0, 4              ; 14: li $v0,4 # Code for syscall:
                                                ; print_string
[00400028] 3c011001  lui $1, 4097 [msg]         ; 15: la $a0, msg # Pointer to string
                                                ; (load the address of msg)
[0040002c] 34240000  ori $4, $1, 0 [msg]
[00400030] 0000000c  syscall                    ; 16: syscall
[00400034] 3402000a  ori $2, $0, 10             ; 17: li $v0,10 # Code for syscall:
                                                ; exit
[00400038] 0000000c  syscall                    ; 18: syscall
```

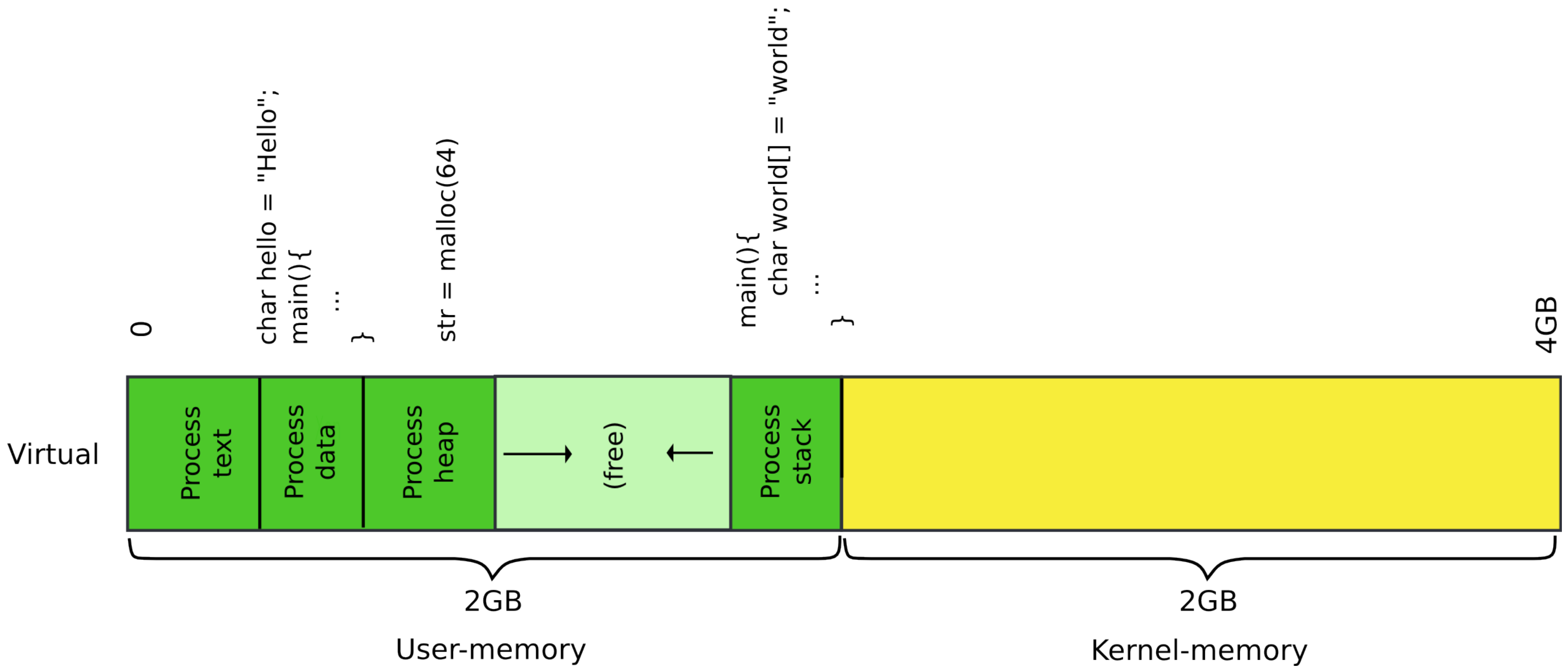
What needs to be relocated?

```
  User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)            ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4         ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4          ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2            ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2          ; 187: addu $a2 $a2 $v0
[00400014] 0c100009  jal 0x00400024 [main]    ; 188: jal main
[00400018] 00000000  nop                      ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10           ; 191: li $v0 10
[00400020] 0000000c  syscall                  ; 192: syscall # syscall 10 (exit)
[00400024] 34020004  ori $2, $0, 4            ; 14: li $v0,4 # Code for syscall:
                                              ; print_string
[00400028] 3c011001  lui $1, 4097 [msg]       ; 15: la $a0, msg # Pointer to string
                                              ; (load the address of msg)
[0040002c] 34240000  ori $4, $1, 0 [msg]
[00400030] 0000000c  syscall                  ; 16: syscall
[00400034] 3402000a  ori $2, $0, 10           ; 17: li $v0,10 # Code for syscall:
                                              ; exit
[00400038] 0000000c  syscall                  ; 18: syscall
```

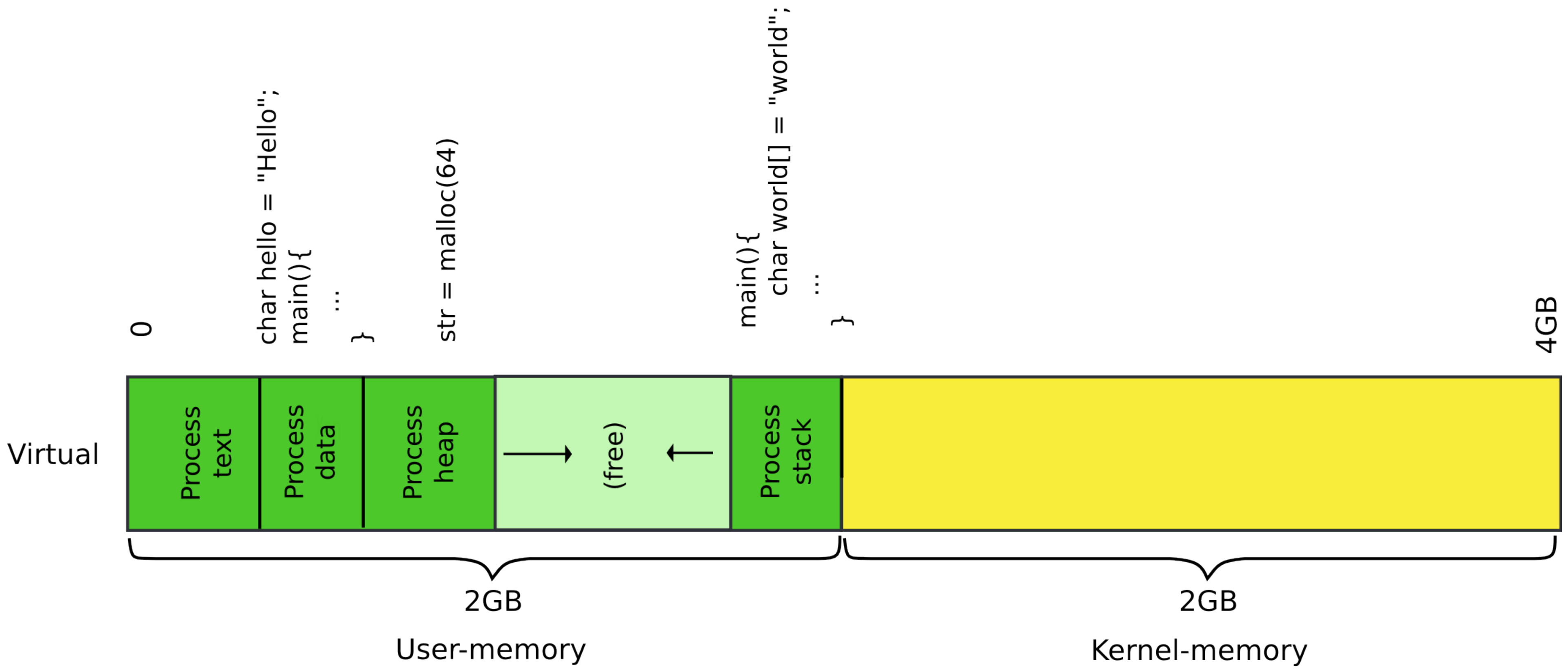
Translation and Startup


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

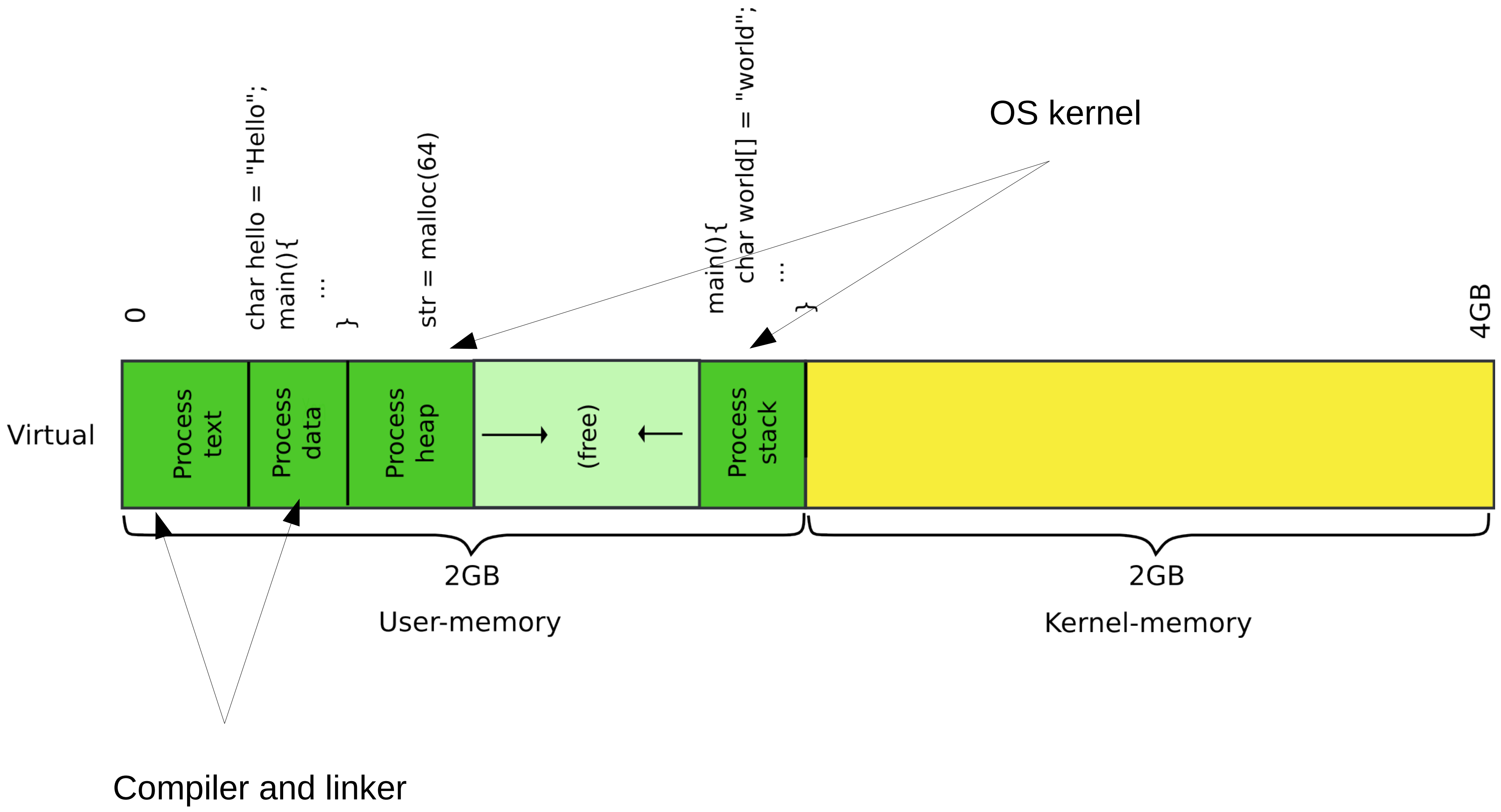
Memory layout of a process



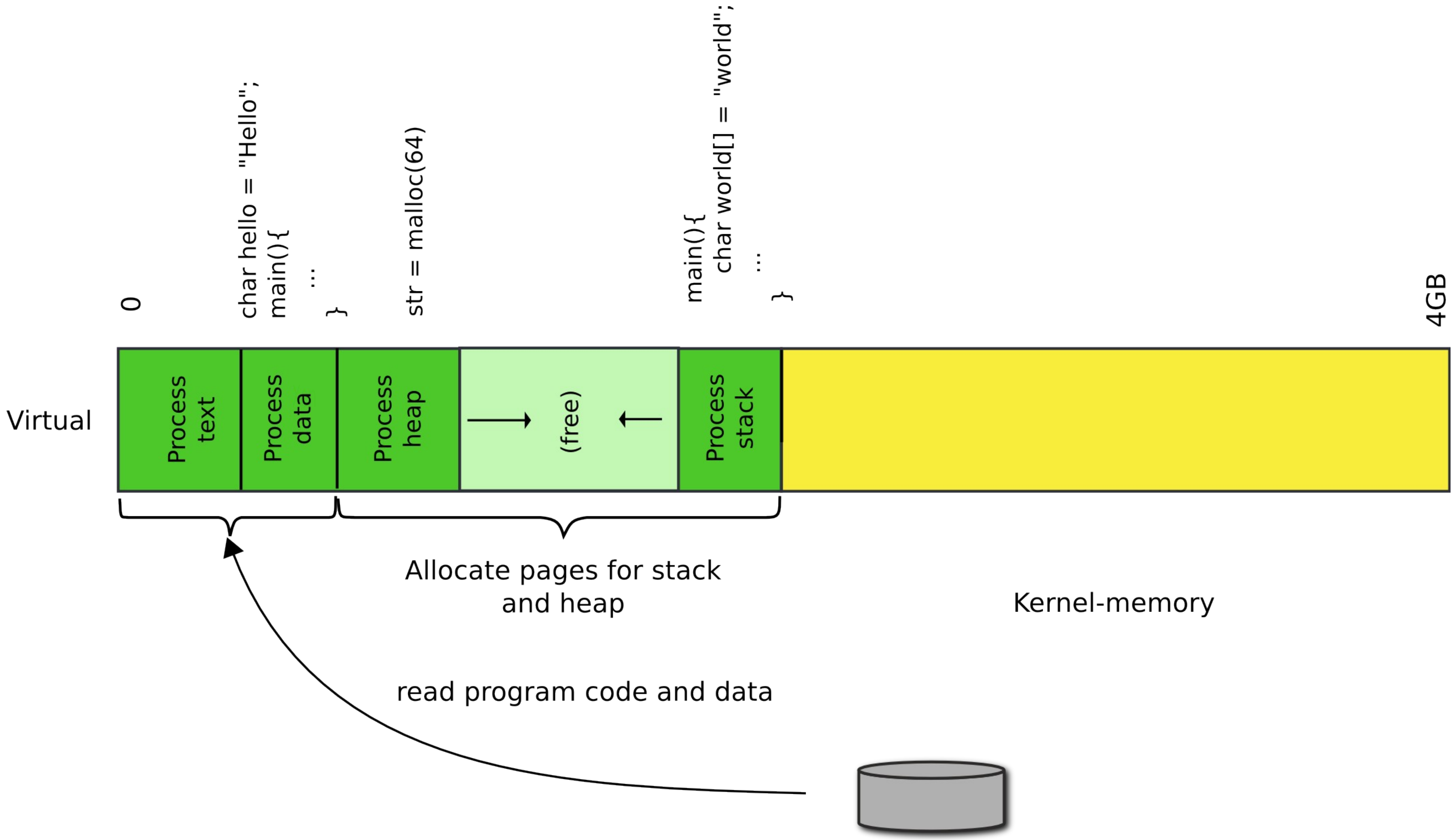
Where do these areas come from?



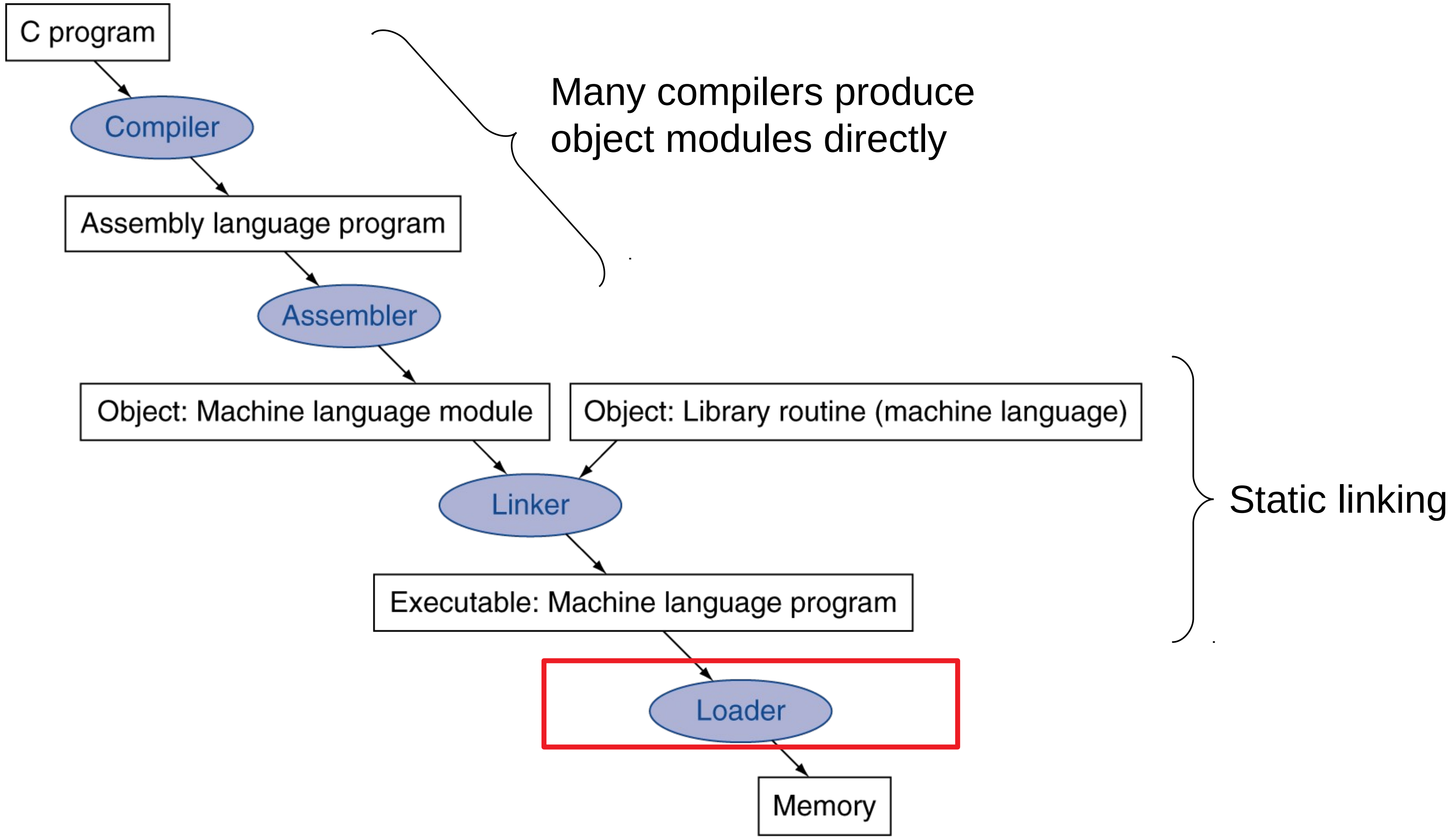
Memory layout of a process



Load program in memory



Translation and Startup


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

Thank you!