CS/EE 3810: Computer Organization

Lecture 9: Midterm Recap

Anton Burtsev
October, 2022

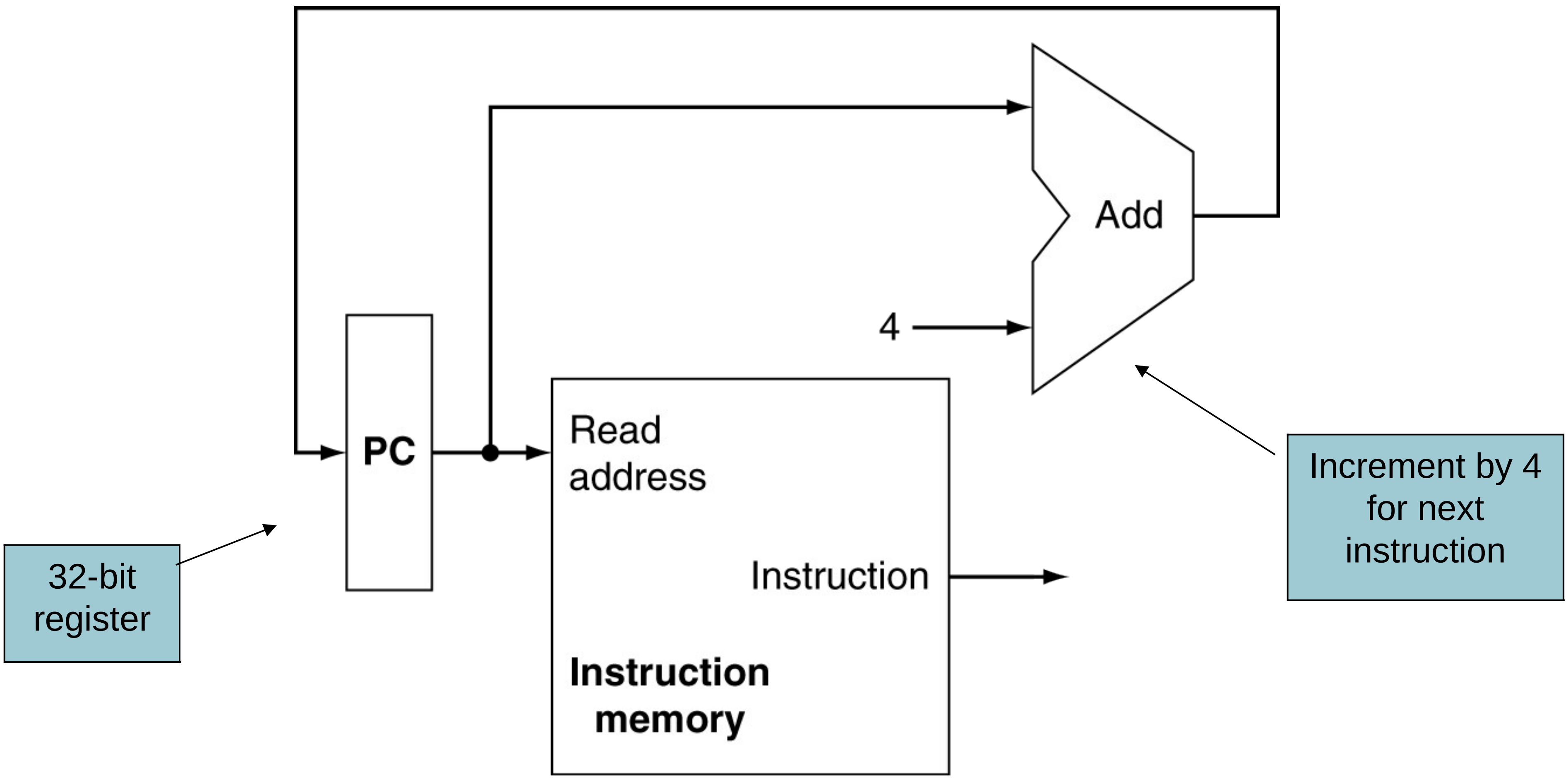
Simple program: String Copy

- 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



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

MIPS Reference Data ①

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]		$0 / 23_{hex}$

Fits on one page

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

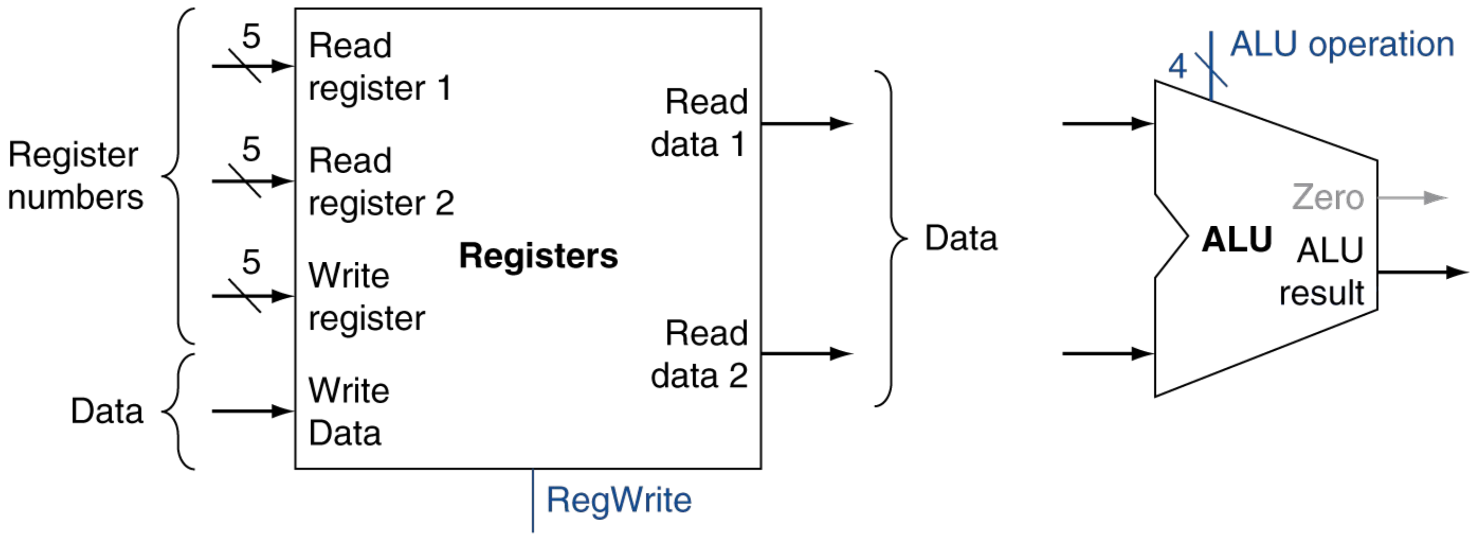
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- J-Type

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

R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

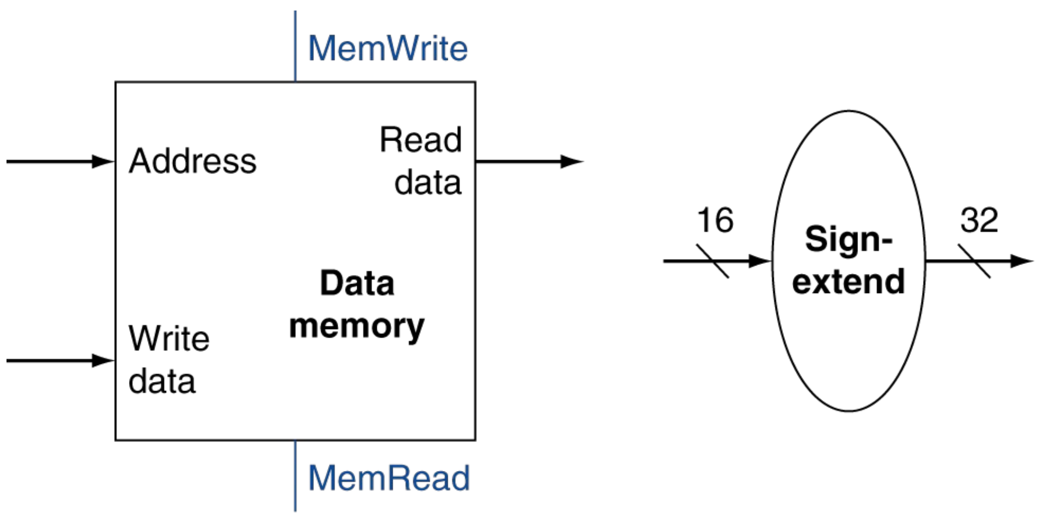


a. Registers

b. ALU

Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

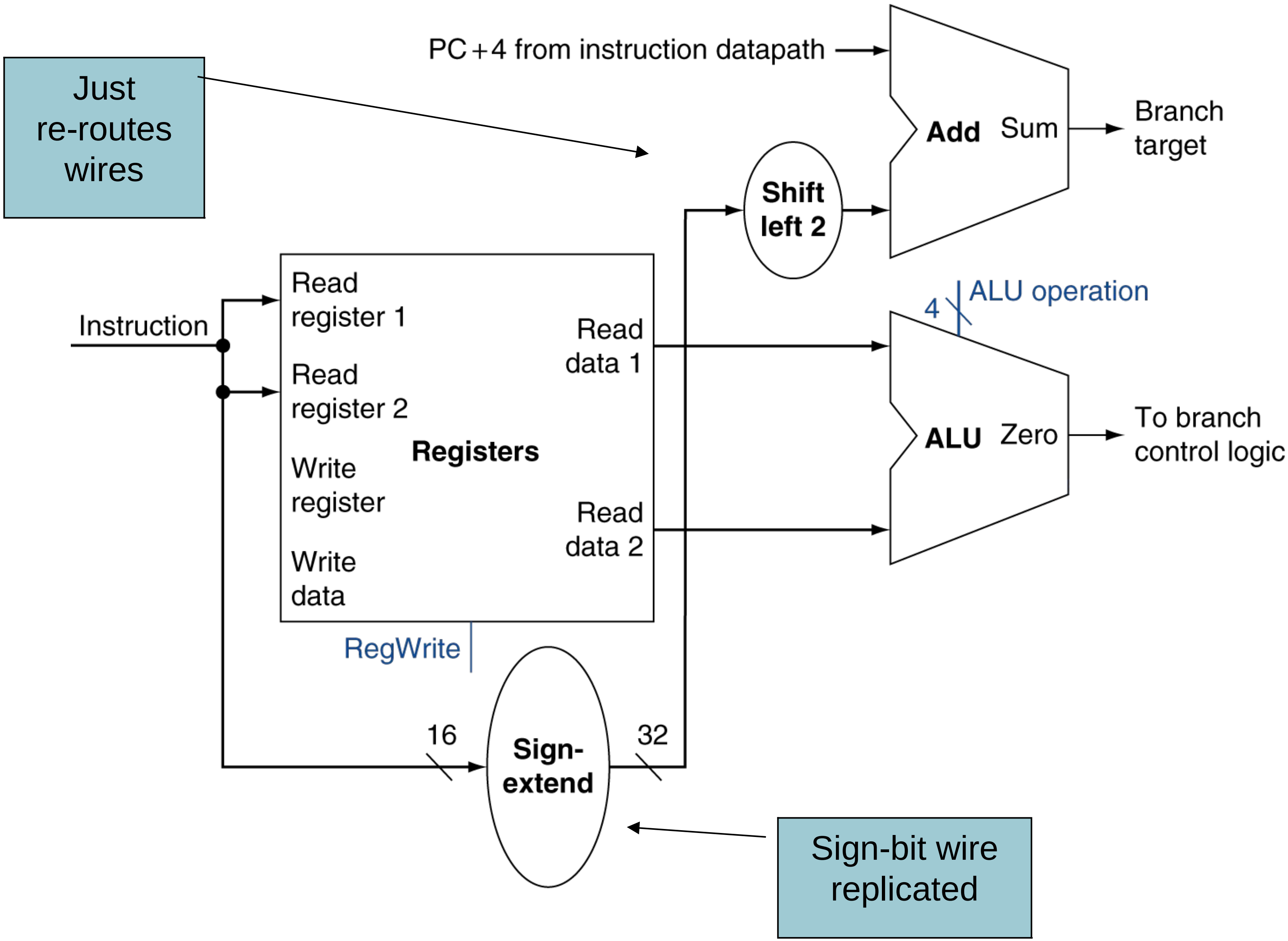


a. Data memory unit   b. Sign extension unit

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

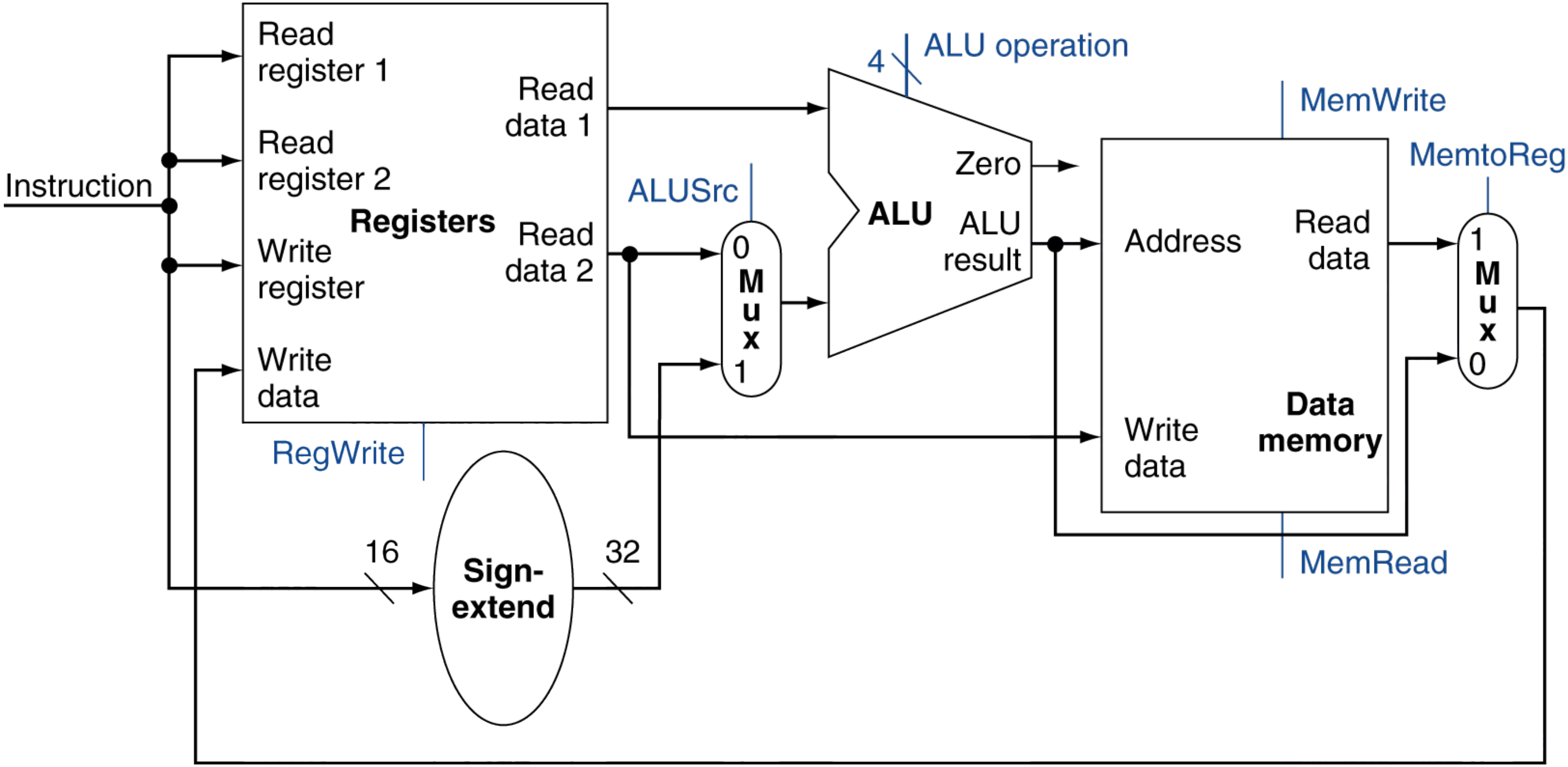
Branch Instructions



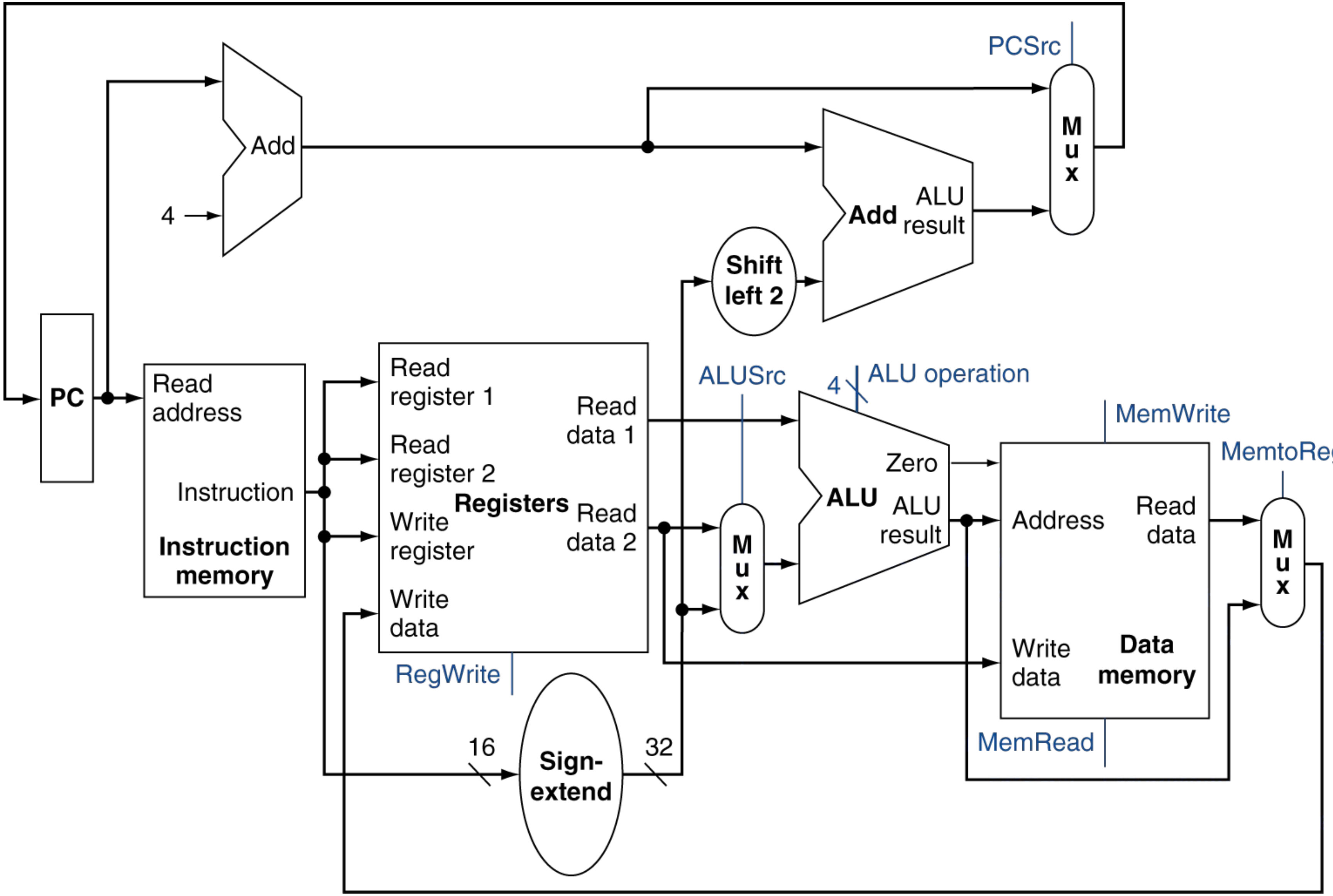
Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

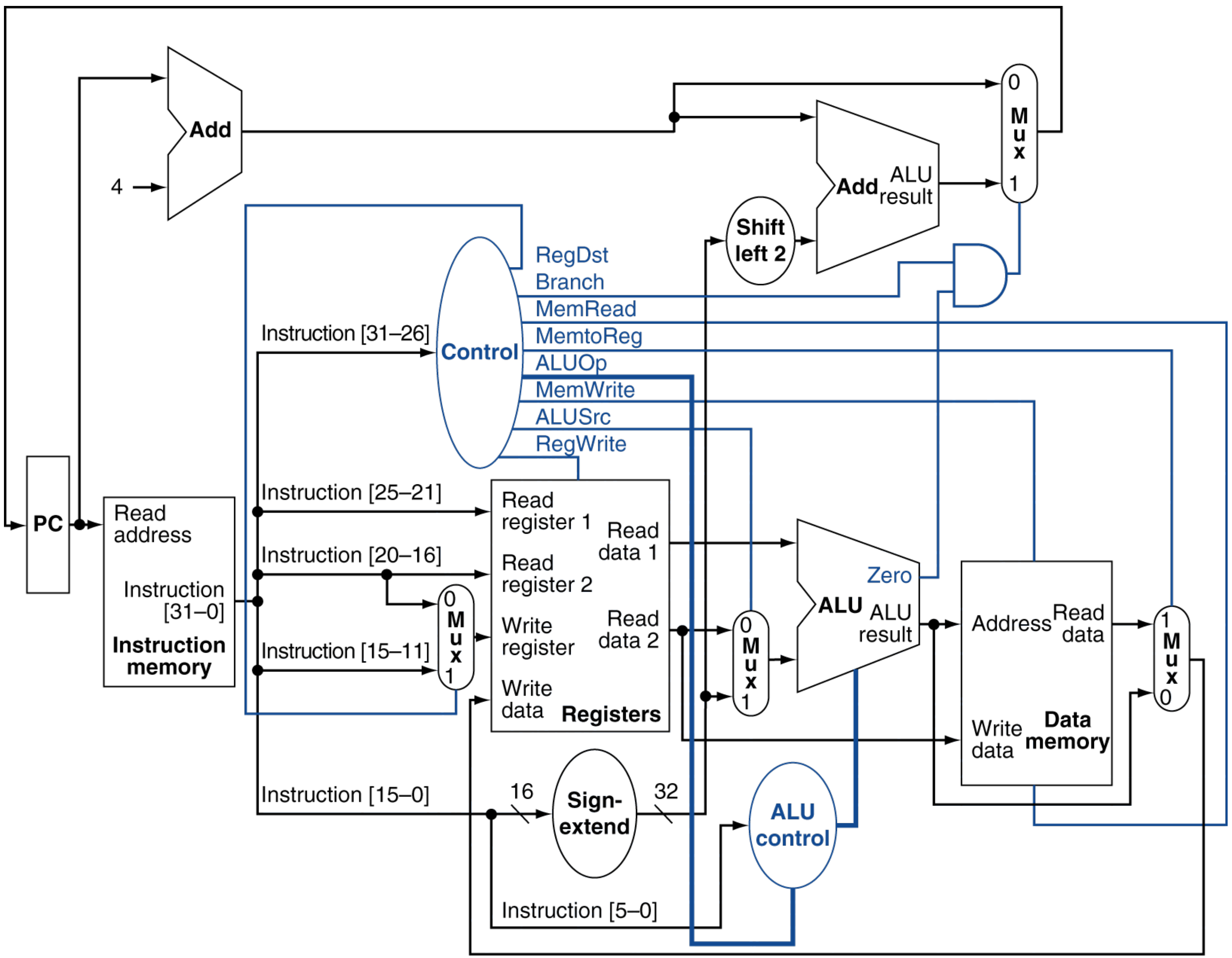
R-Type/Load/Store Datapath



Full Datapath



Datapath With Control



ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

The Main Control Unit

- Control signals derived from instruction

R-type

0	rs	rt	rd	shamt	funct
31:26	25:21	20:16	15:11	10:6	5:0

Load/ Store

35 or 43	rs	rt	address
31:26	25:21	20:16	15:0

Branch

4	rs	rt	address
31:26	25:21	20:16	15:0

opcode

always read

read, except for load

write for R-type and load

sign-extend and add

The Main Control Unit

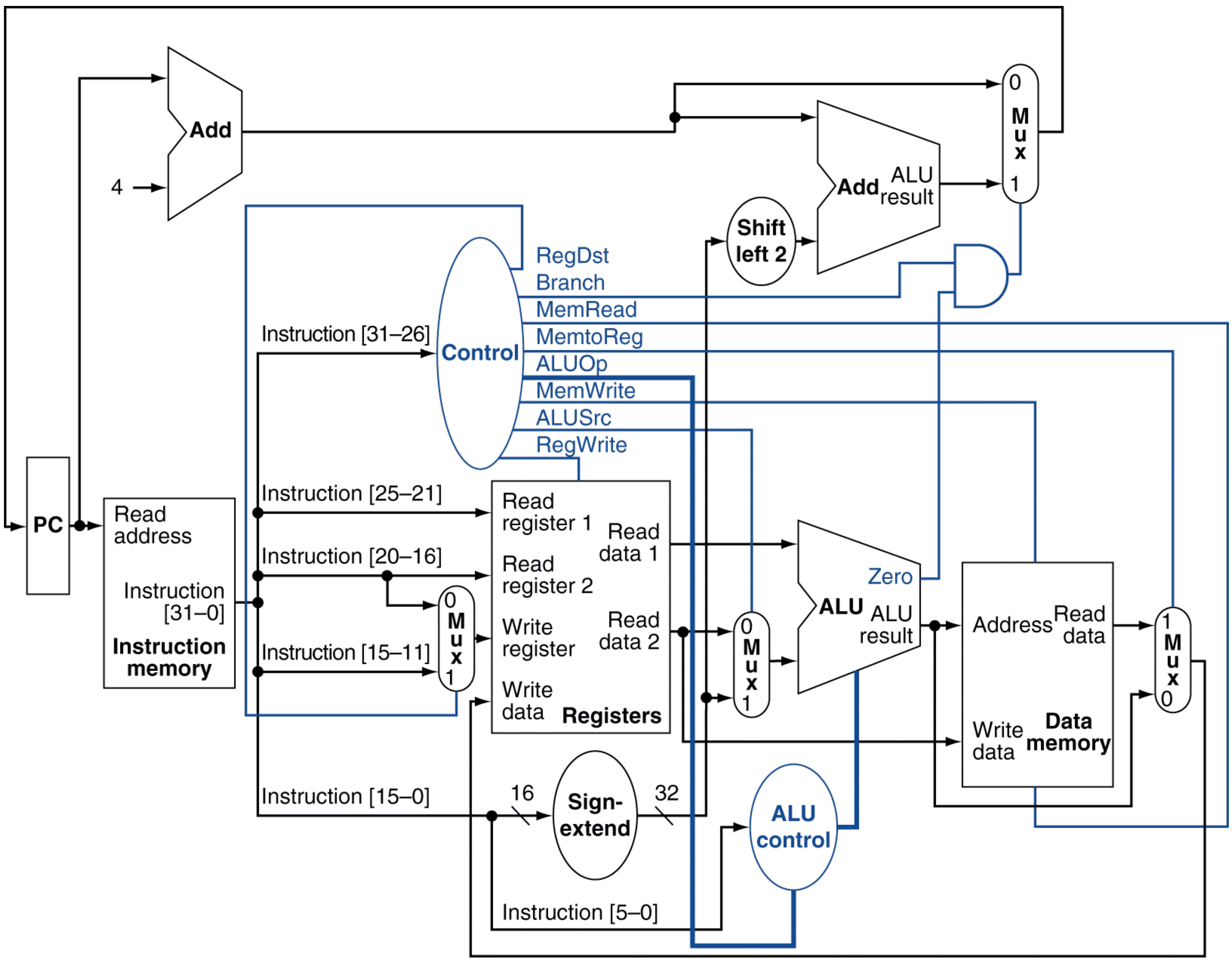
Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

FIGURE 4.18 The setting of the control lines is completely determined by the opcode fields of the instruction. The first row of the table corresponds to the R-format instructions (`add`, `sub`, `AND`, `OR`, and `slt`). For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set. Furthermore, an R-type instruction writes a register (Reg-Write = 1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high. The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field. The second and third rows of this table give the control signal settings for `lw` and `sw`. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register. The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality. Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

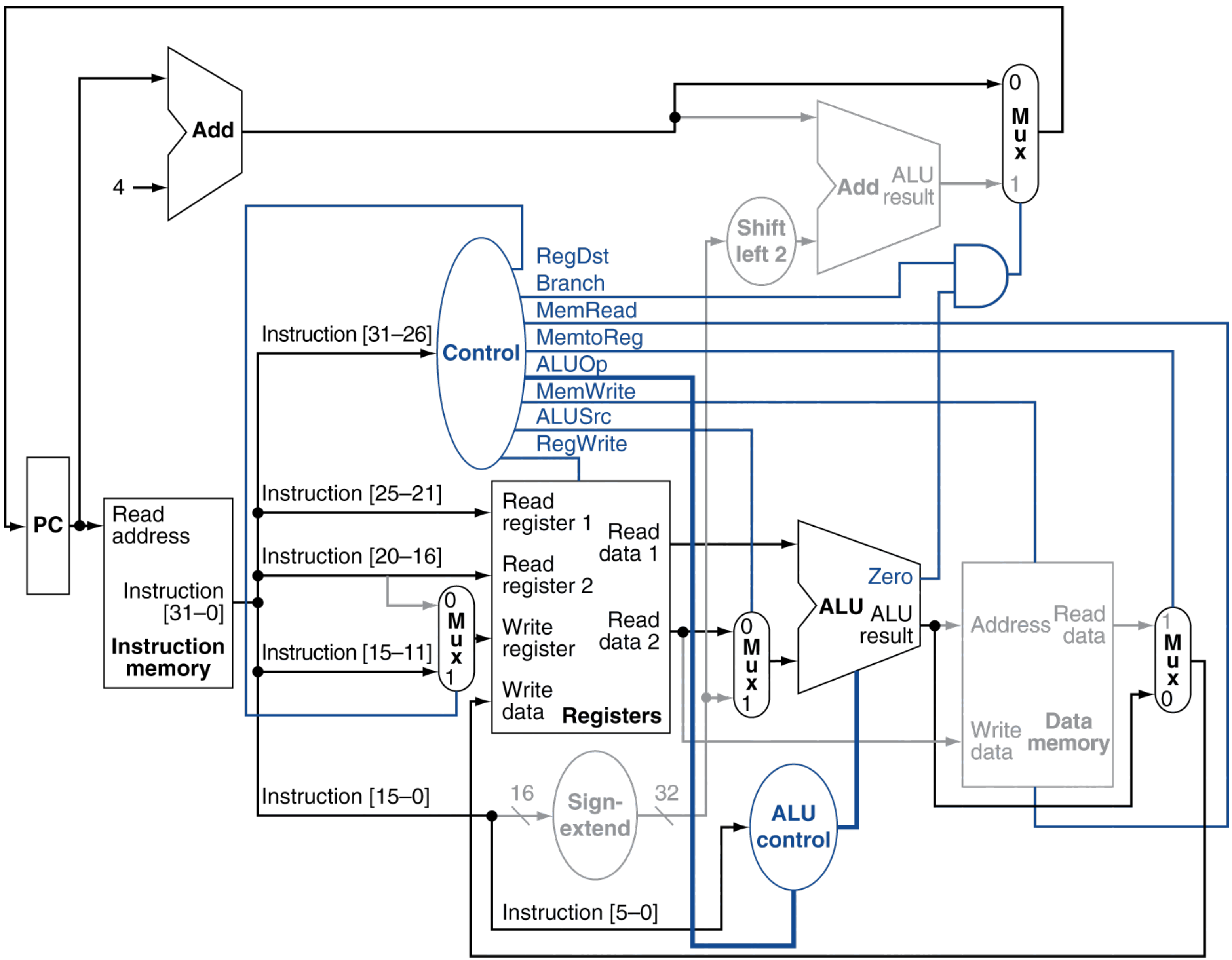
Input or output	Signal name	R-format	`lw`	`sw`	`beq`
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

FIGURE 4.22 The control function for the simple single-cycle implementation is completely specified by this truth table. The top half of the table gives the combinations of input signals that correspond to the four opcodes, one per column, that determine the control output settings. (Remember that Op [5:0] corresponds to bits 31:26 of the instruction, which is the op field.) The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output RegWrite is asserted for two different combinations of the inputs. If we consider only the four opcodes shown in this table, then we can simplify the truth table by using don't cares in the input portion. For example, we can detect an R-format instruction with the expression $\overline{\text{Op5}} \cdot \overline{\text{Op2}}$, since this is sufficient to distinguish the R-format instructions from `lw`, `sw`, and `beq`. We do not take advantage of this simplification, since the rest of the MIPS opcodes are used in a full implementation.

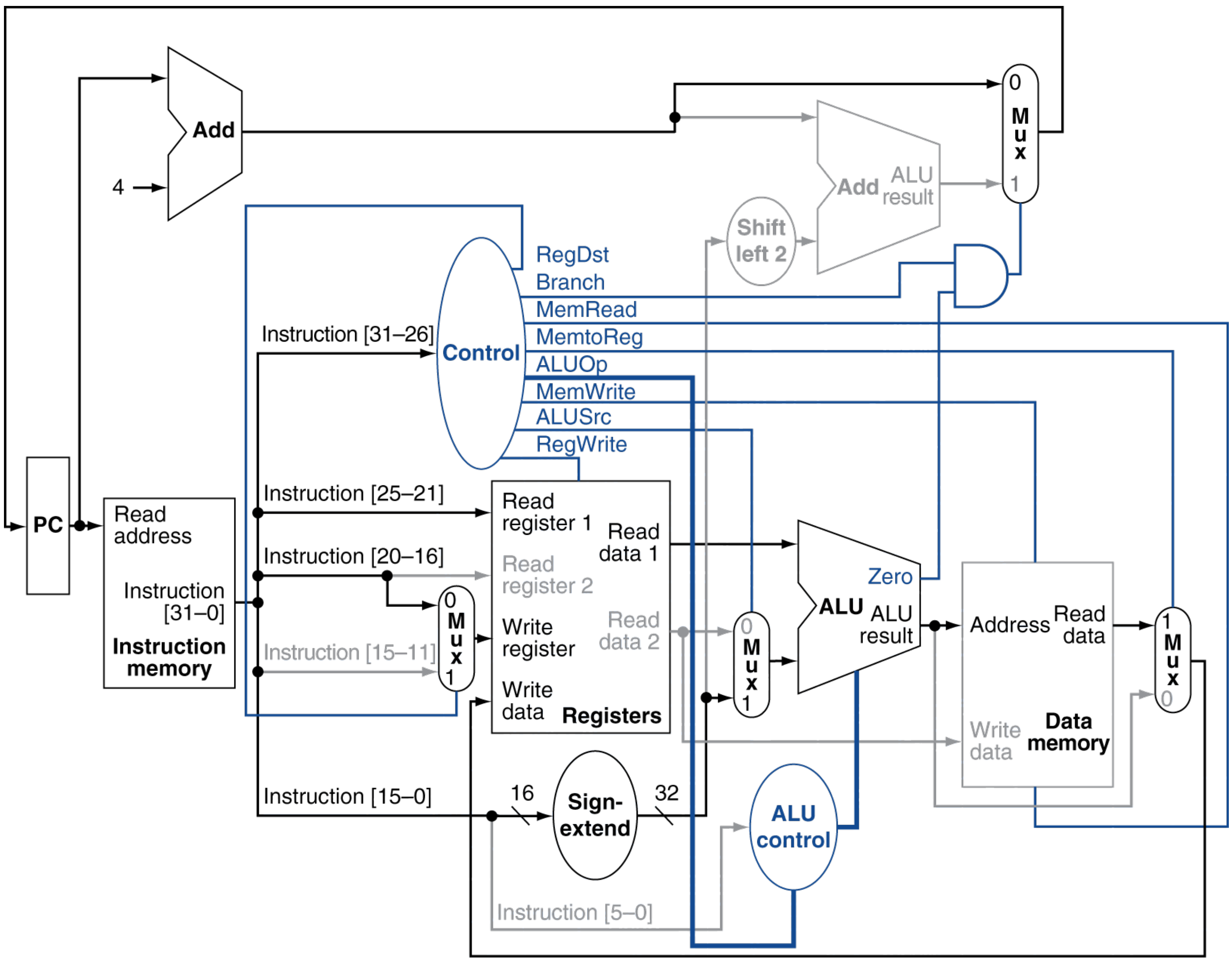
Datapath With Control



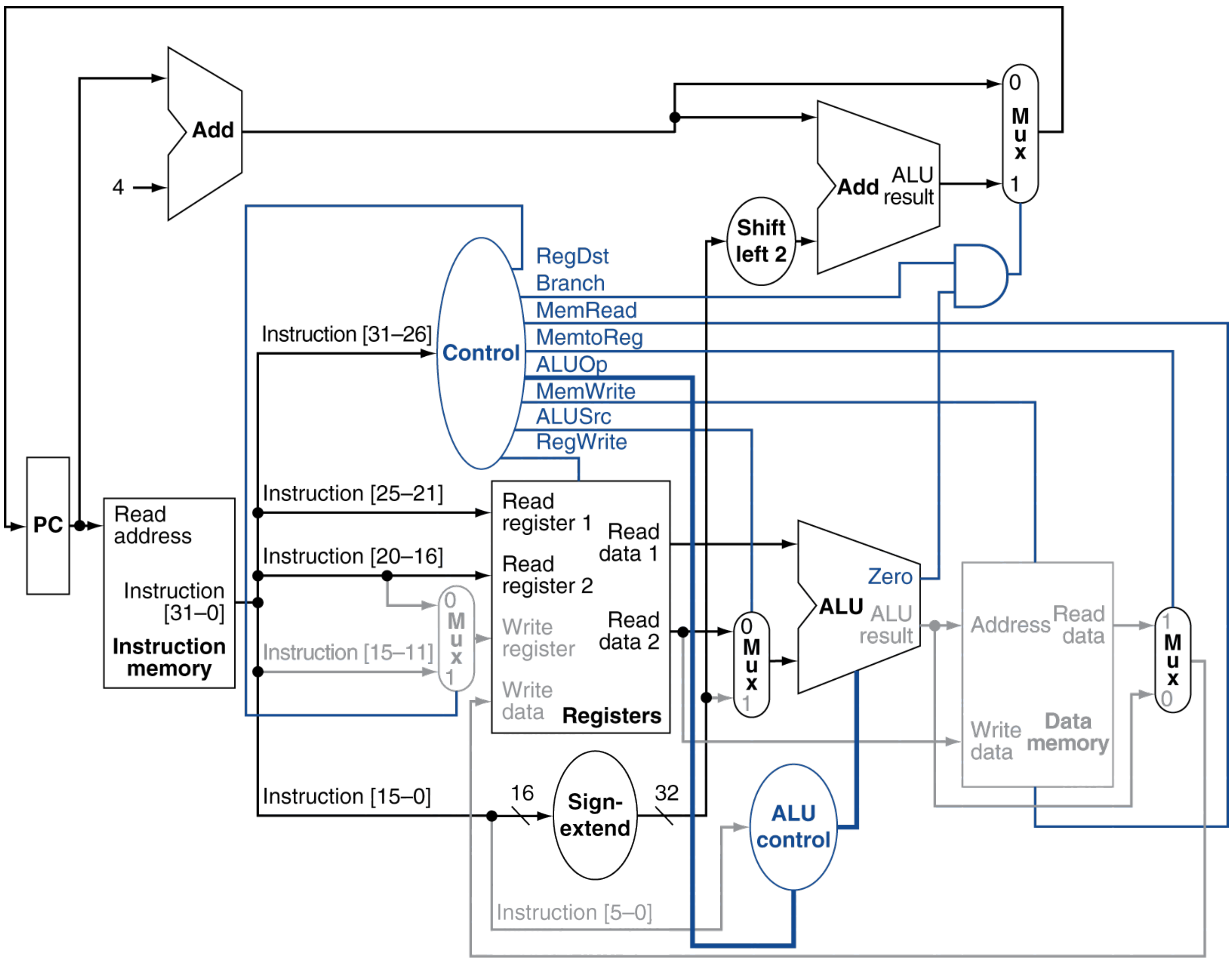
R-Type Instruction



Load Instruction



Branch-on-Equal Instruction

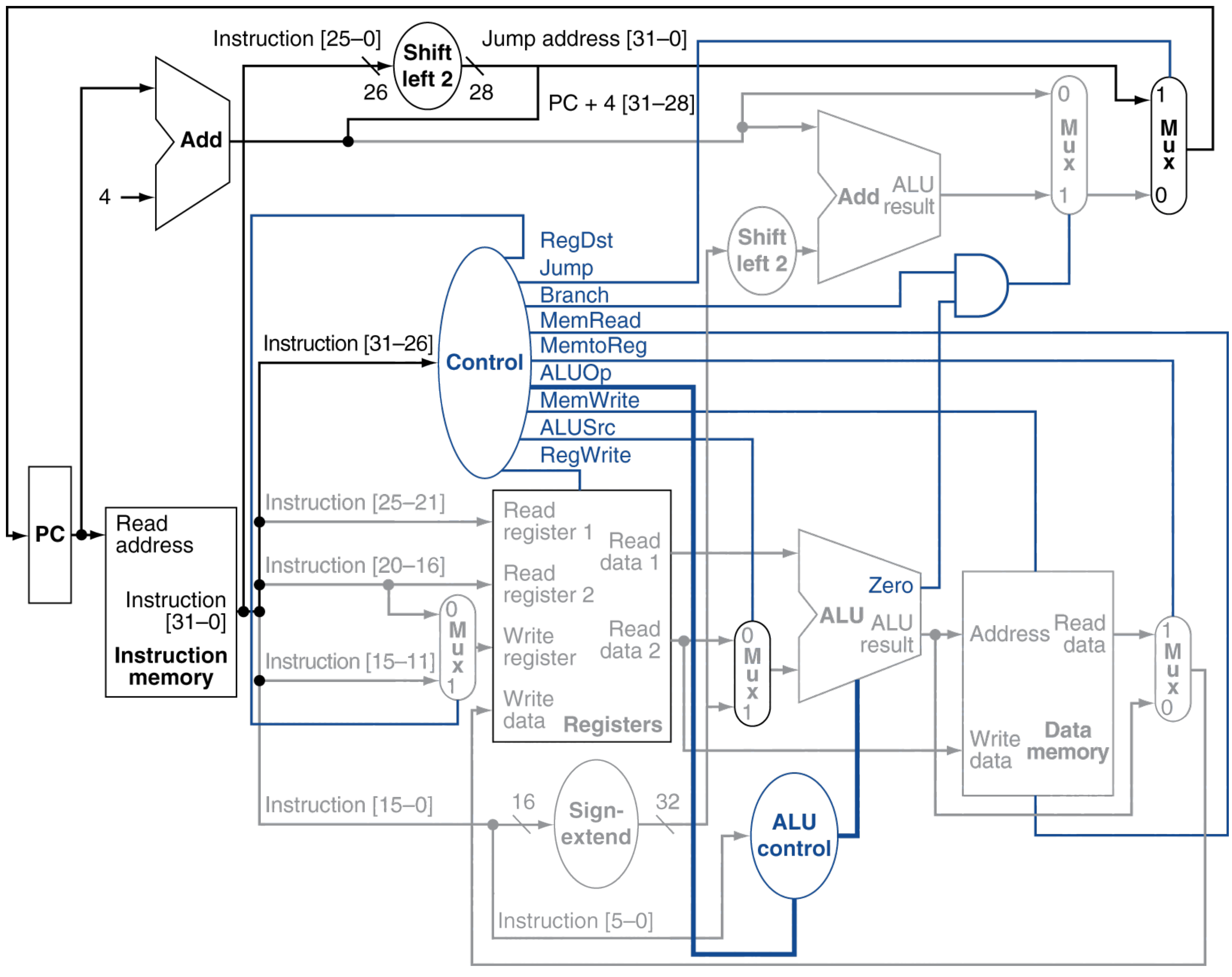


Implementing Jumps

Jump	2	address
	31:26	25:0

- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

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

High-level language program (in C)

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

Compiler

Assembly language program (for MIPS)

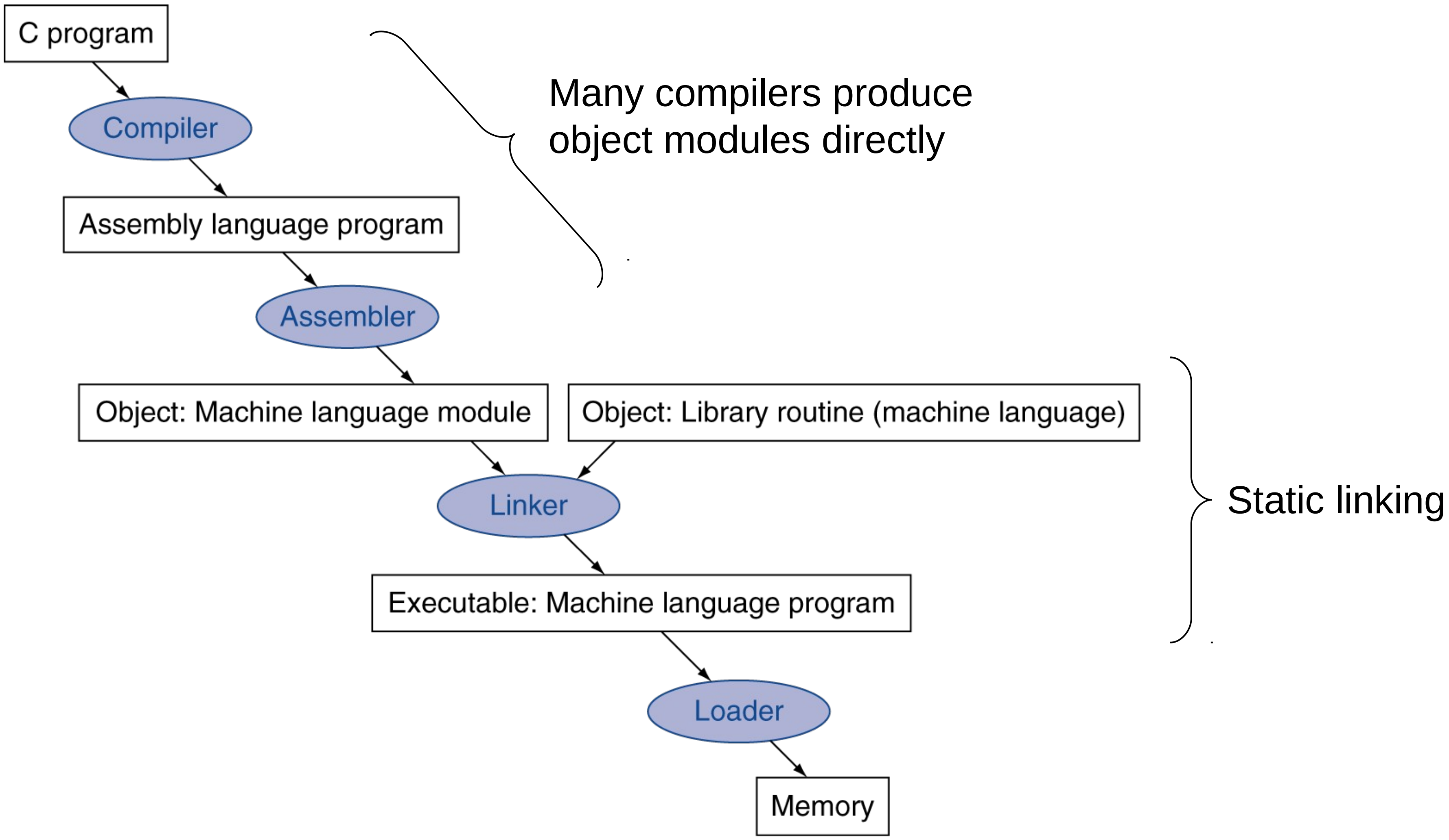
```
swap:
      multi $2, $5,4
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```

Assembler

Binary machine language program (for MIPS)

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

Translation and Startup



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
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $s0, 0($sp)
    addi $sp, $sp, 4
    jr   $ra
```

Code	Comment
addi $sp, $sp, -4 sw $s0, 0($sp)	Save $s0 on stack

add $t0, $a0, $a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
	Procedure body

add $v0, $s0, $zero	Result
lw $s0, 0($sp) addi $sp, $sp, 4	Restore $s0
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

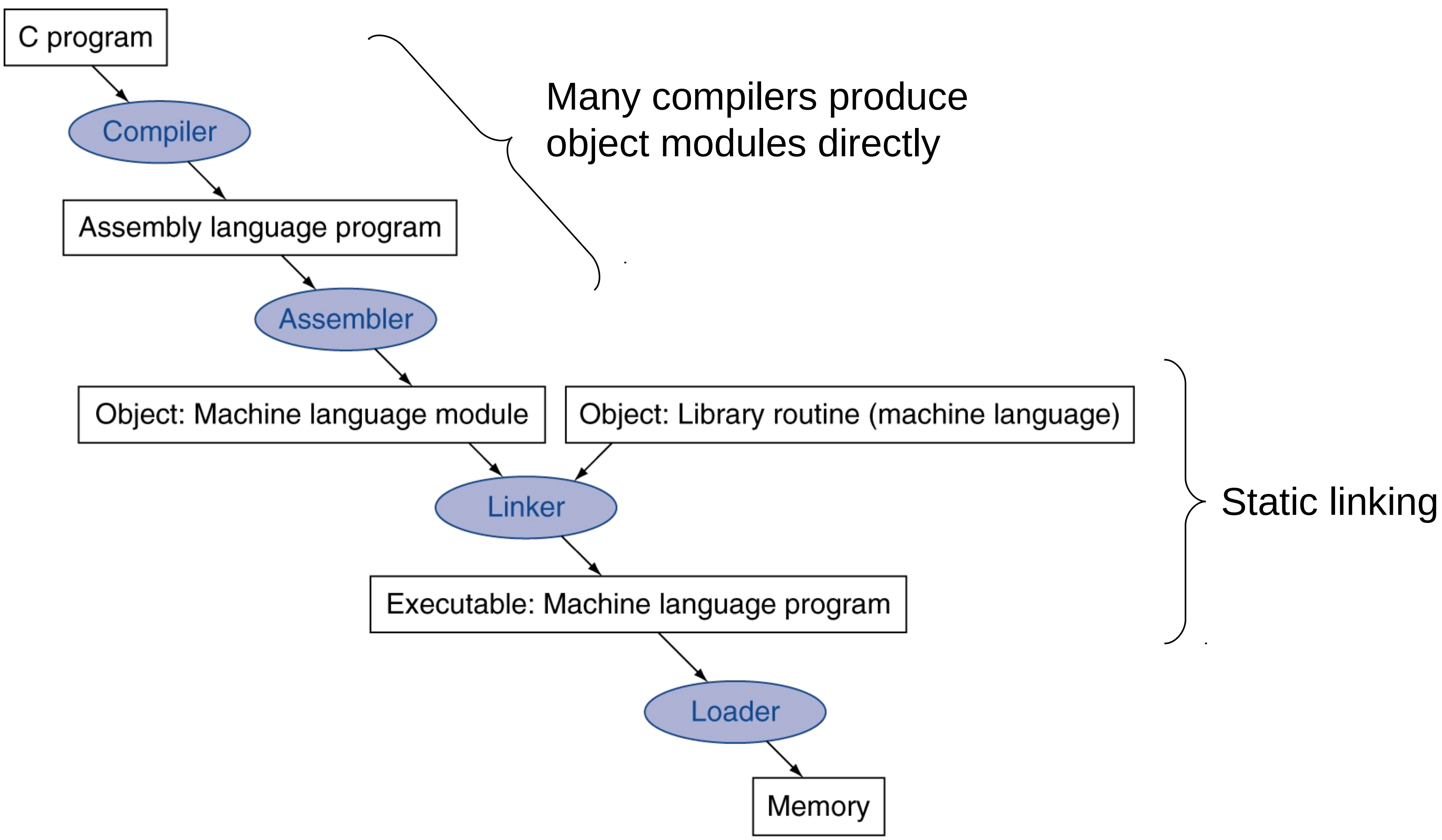
- MIPS code:

```
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
```

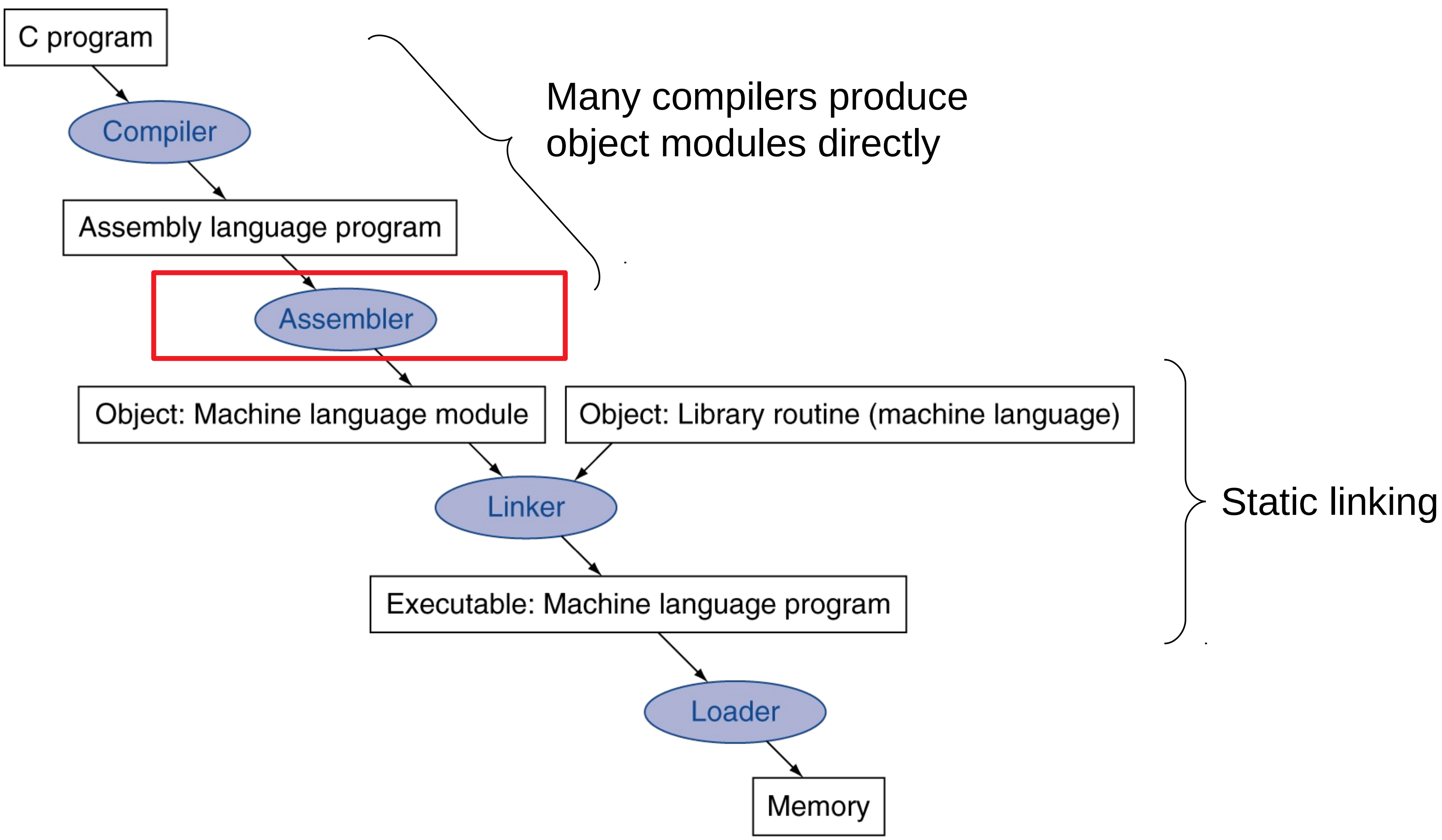
Calling convention (again)

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

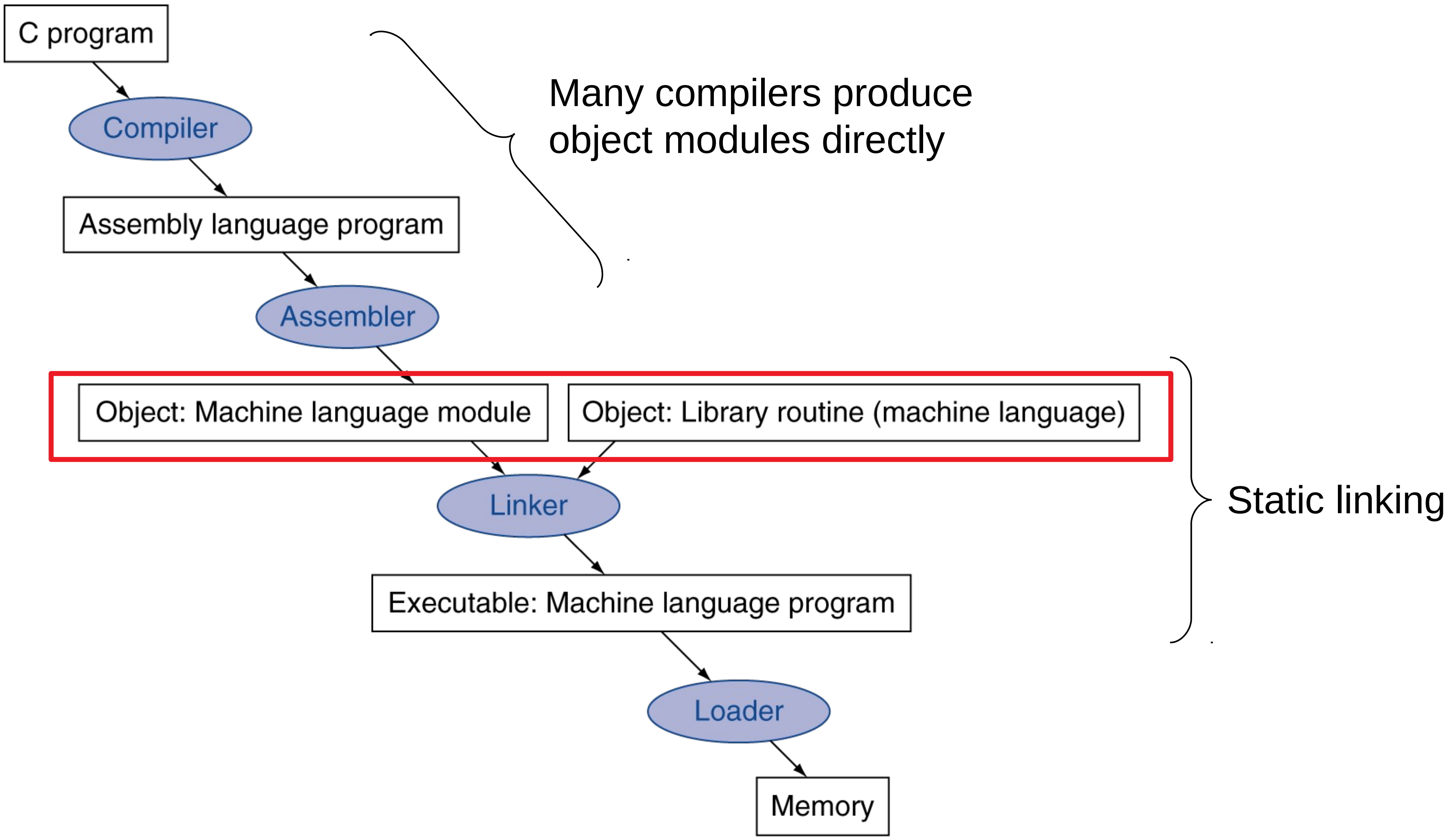
Translation and Startup



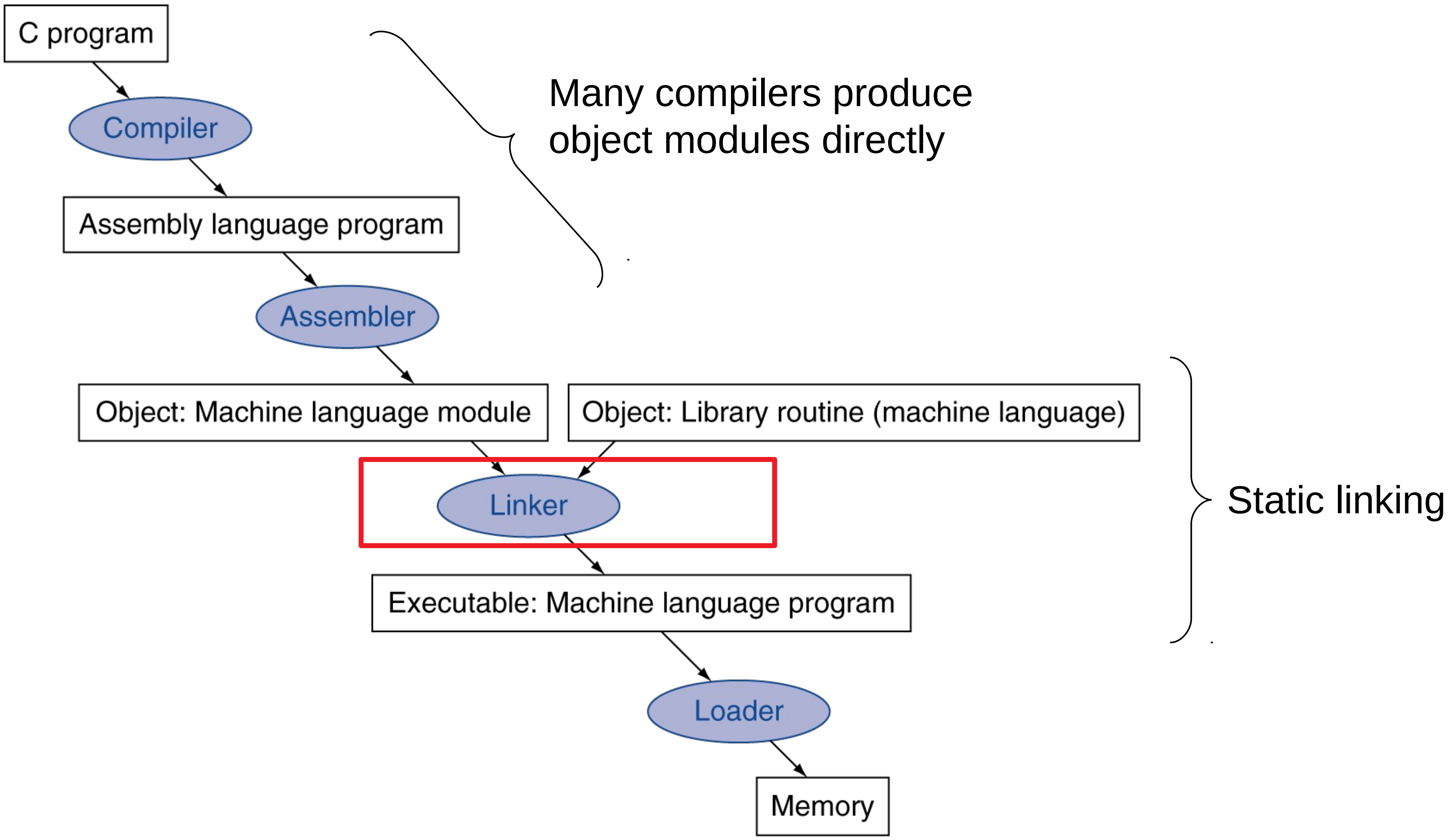
Translation and Startup

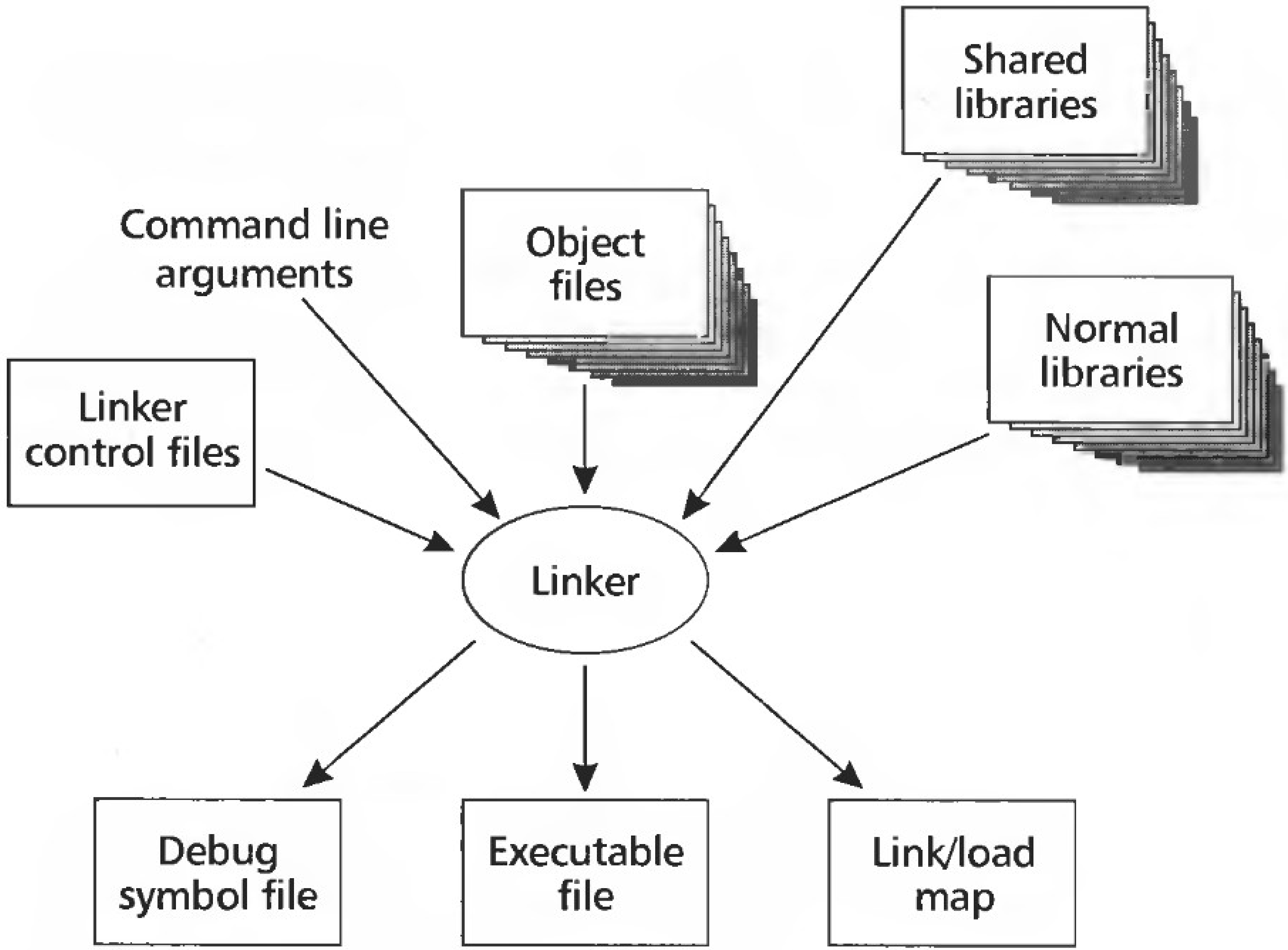


Translation and Startup



Translation and Startup




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

```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;
}
```

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;
}
```

- 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

```
#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"
```

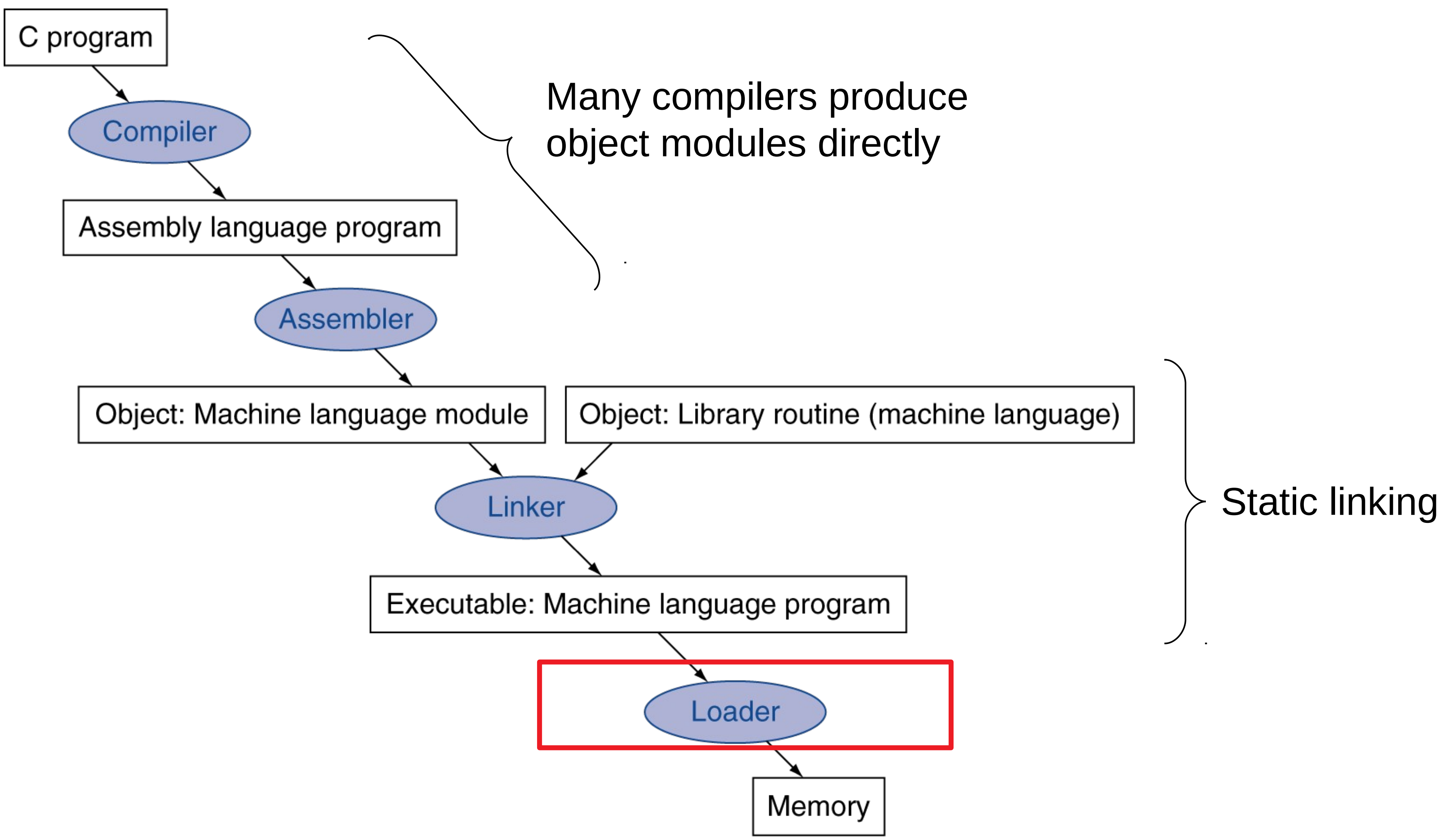
What needs to be relocated?

```
  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
```

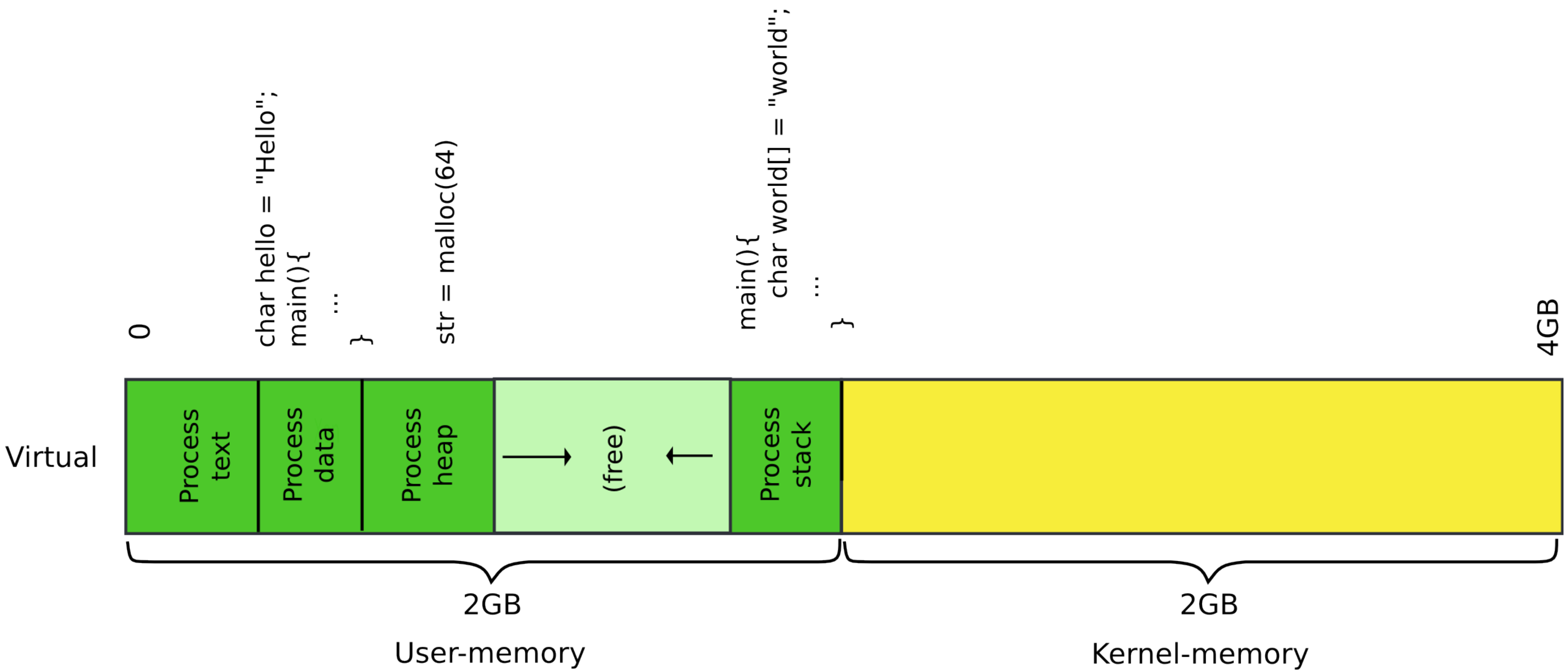
What needs to be relocated?

```
  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
```

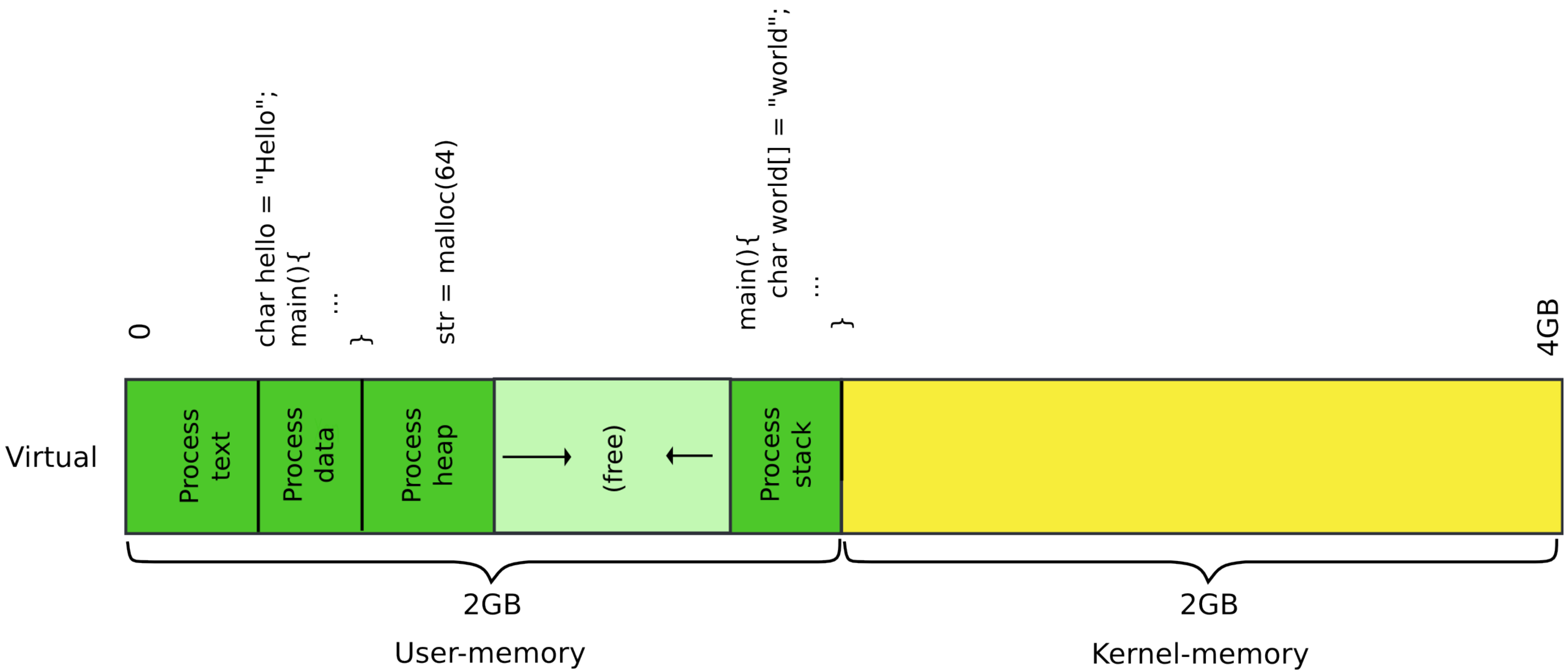
Translation and Startup



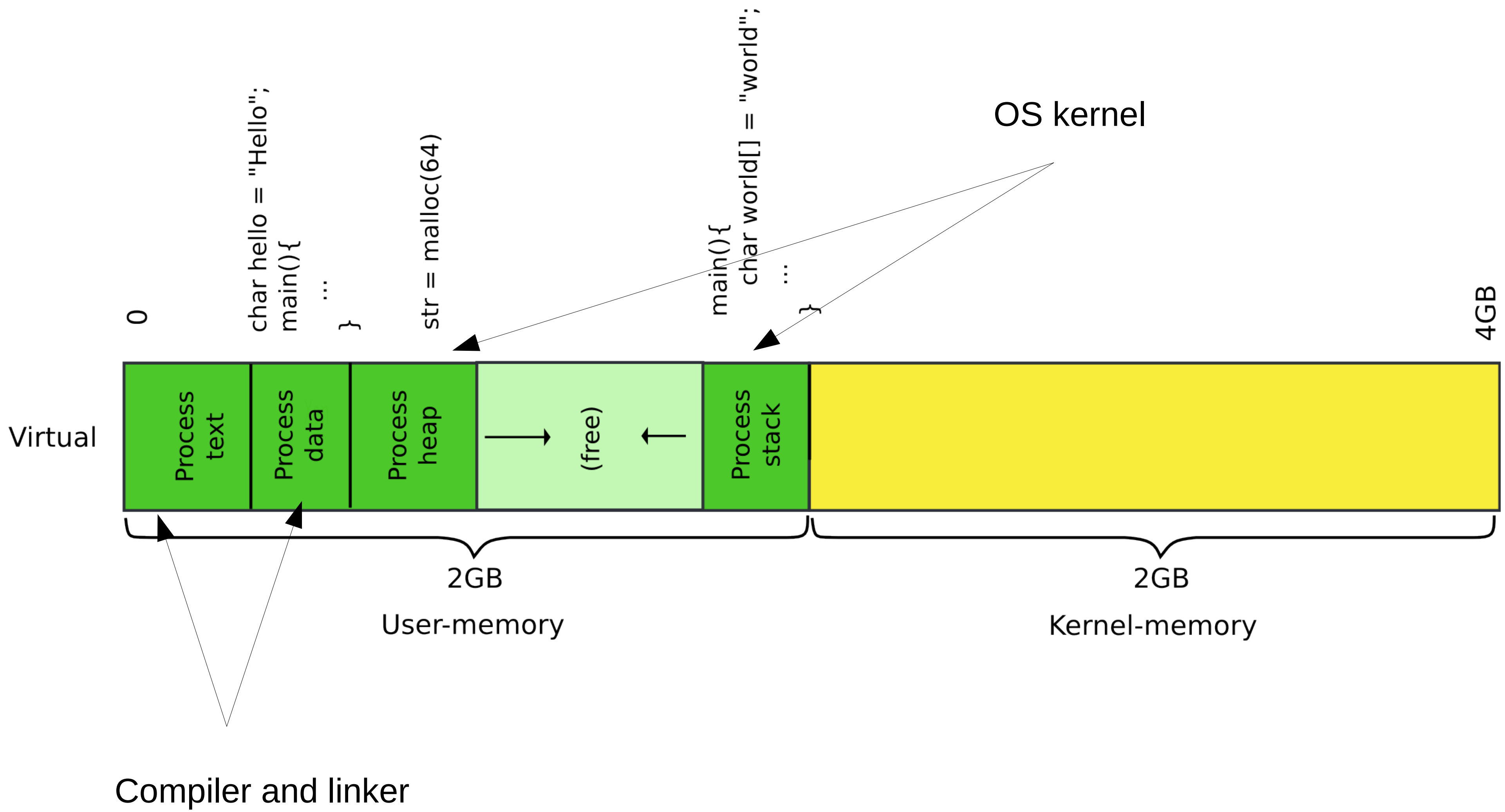
Memory layout of a process



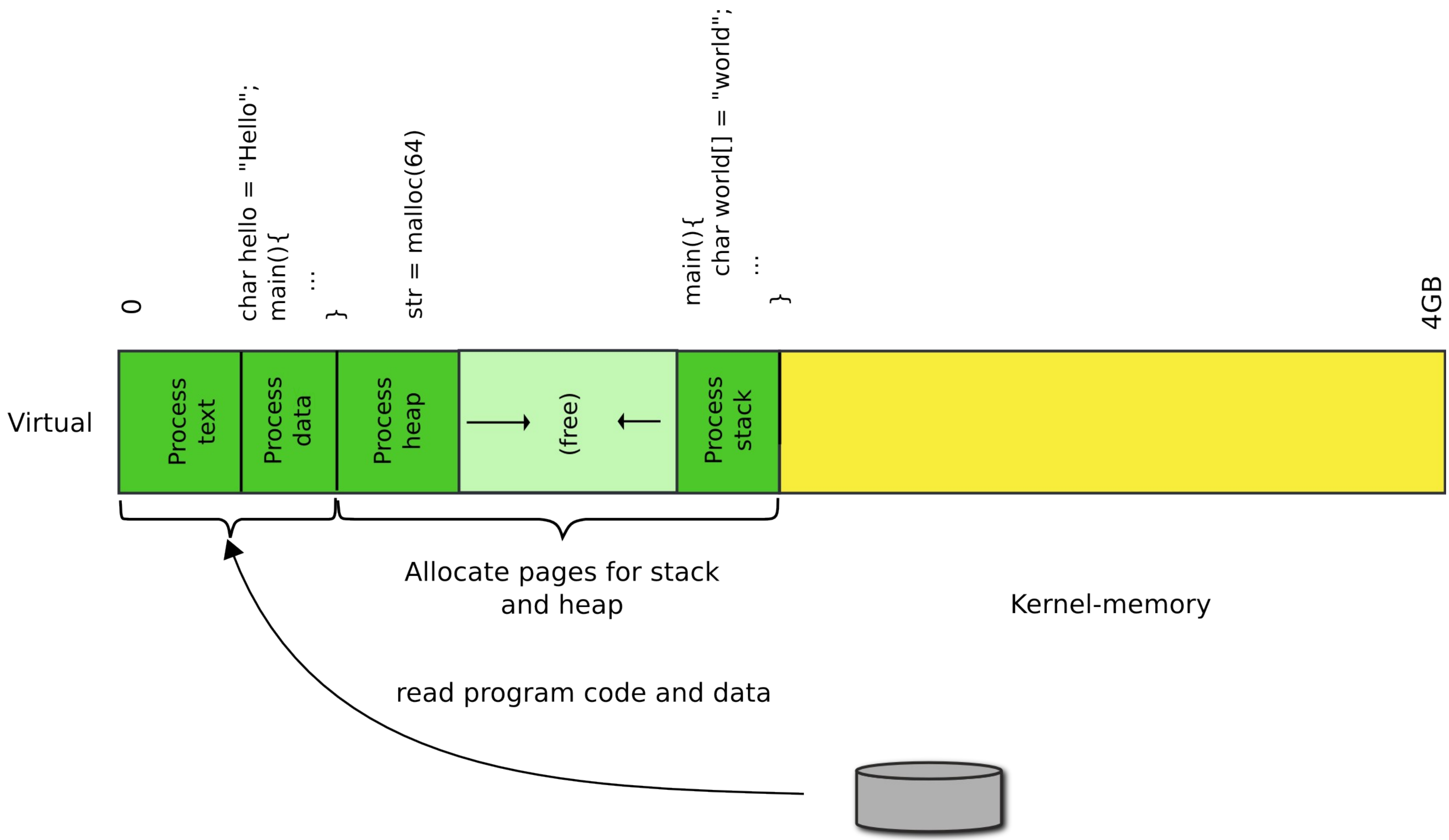
Where do these areas come from?



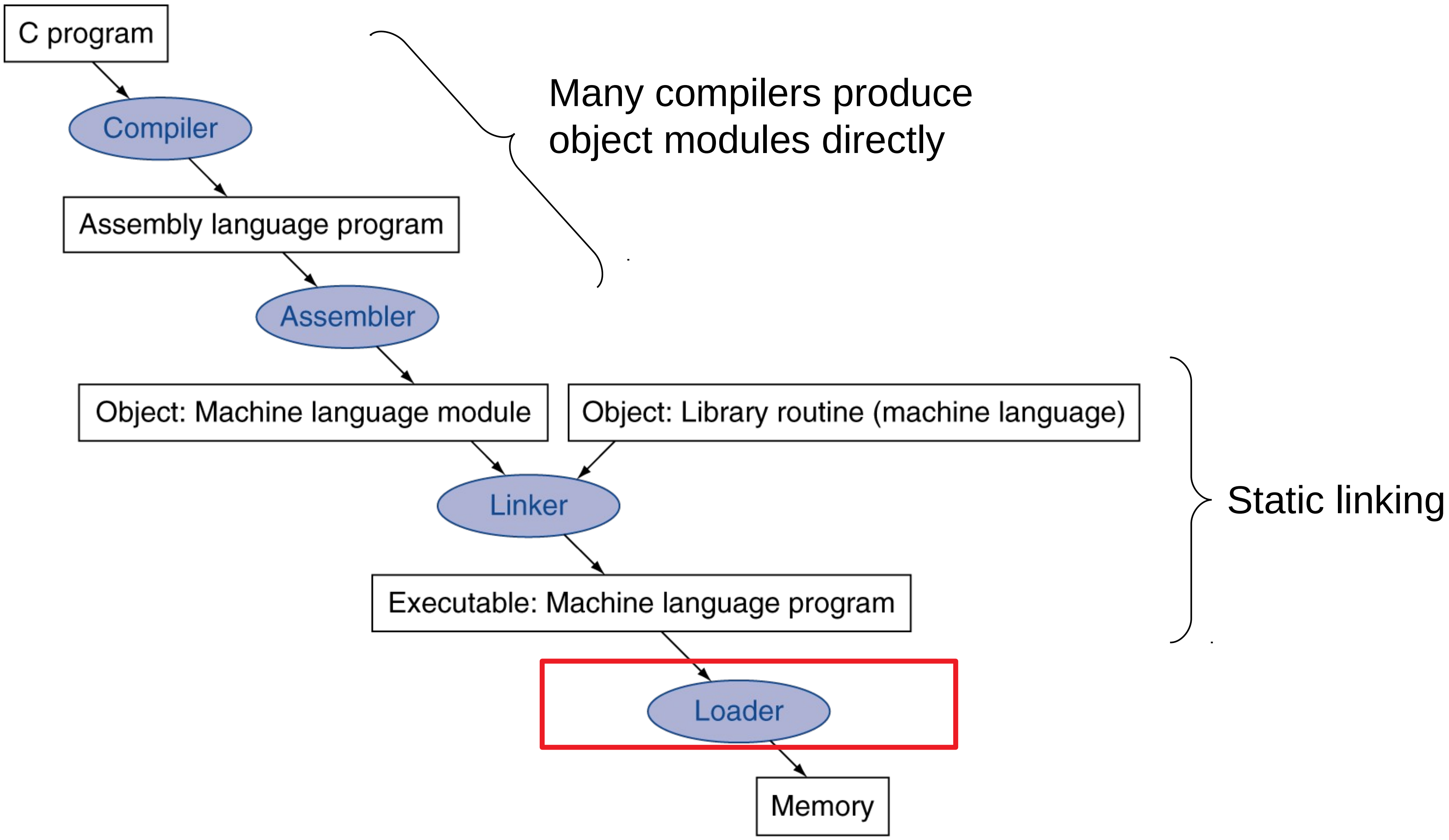
Memory layout of a process



Load program in memory



Translation and Startup



Thank you!