CS/ECE 3810: Computer Organization

Lecture 4: MIPS instruction set

Anton Burtsev

September, 2022

Instruction Set

- Understanding the language of the hardware is key to understanding the hardware/software interface

- A program (in say, C) is compiled into an executable that is composed of machine instructions – this executable must also run on future machines – for example, each Intel processor reads in the same x86 instructions, but each processor handles instructions differently

- Java programs are converted into portable bytecode that is converted into machine instructions during execution (just-in-time compilation)

- What are important design principles when defining the instruction set architecture (ISA)?

Instruction Set

- Important design principles when defining the instruction set architecture (ISA):
  - keep the hardware simple – the chip must only implement basic primitives and run fast
  - keep the instructions regular – simplifies the decoding/scheduling of instructions

We will later discuss RISC vs CISC

A Basic MIPS Instruction

C code: a = b + c ;

Assembly code: (human-friendly machine instructions)

```
add   a, b, c      #  a is the sum of b and c
```

A Basic MIPS Instruction

C code: a = b + c ;

Assembly code: (human-friendly machine instructions)

```
add   a, b, c      #  a is the sum of b and c
```

Machine code: (hardware-friendly machine instructions)

```
00000010001100100100000000100000
```

Translate the following C code into assembly code:

a = b + c + d + e;

Example

C code   a = b + c + d + e;

translates into the following assembly code:

Example

C code   a = b + c + d + e;

translates into the following assembly code:

```
add  a, b, c                add  a, b, c
add  a, a, d        or      add  f, d, e
add  a, a, e                add  a, a, f
```

- Instructions are simple: fixed number of operands (unlike C)
- A single line of C code is converted into multiple lines of assembly code
- Some sequences are better than others… the second sequence needs one more (temporary) variable f

Subtract Example

C code   f = (g + h) – (i + j);

Assembly code translation with only add and sub instructions:

Subtract Example

C code   f = (g + h) – (i + j);

translates into the following assembly code:

```
add  t0, g, h               add  f, g, h
add  t1,  i, j        or     sub   f, f, i
sub  f,   t0, t1            sub   f, f, j
```

Operands

- In C, each “variable” is a location in memory

- In hardware, each memory access is expensive – if variable *a* is accessed repeatedly, it helps to bring the variable into an on-chip scratchpad and operate on the scratchpad (registers)

- To simplify the instructions, we require that each instruction (add, sub) only operate on registers

- Note: the number of operands (variables) in a C program is very large; the number of operands in assembly is fixed… there can be only so many scratchpad registers

Registers

- The MIPS ISA has 32 registers (x86 has 8 registers) – Why not more? Why not less?

- Each register is 32 bits wide (modern 64-bit architectures have 64-bit wide registers)

- A 32-bit entity (4 bytes) is referred to as a word

- To make the code more readable, registers are partitioned as $s0-$s7 (C/Java variables), $t0-$t9 (temporary variables)…

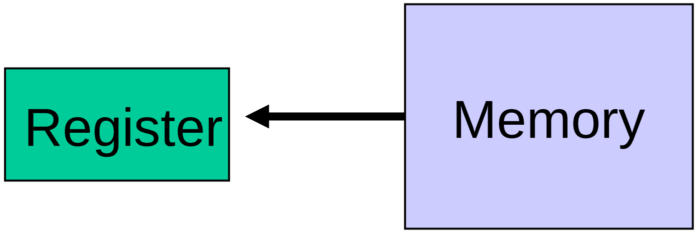
Binary Stuff

- 8 bits = 1 Byte, also written as 8b = 1B
- 1 word = 32 bits = 4B
- 1KB = 1024 B = 2^{10} B
- 1MB = 1024 x 1024 B = 2^{20} B
- 1GB = 1024 x 1024 x 1024 B = 2^{30} B
- A 32-bit memory address refers to a number between 0 and 2^{32} – 1, i.e., it identifies a byte in a 4GB memory

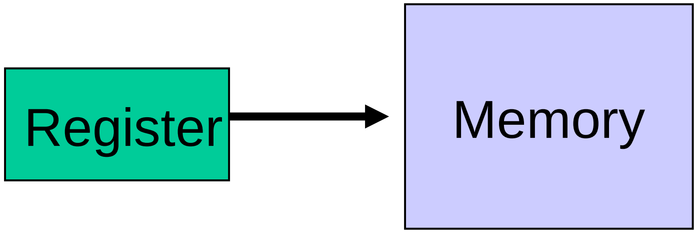
Memory Operands

- Values must be fetched from memory before (add and sub) instructions can operate on them

Load word
lw $t0, memory-address



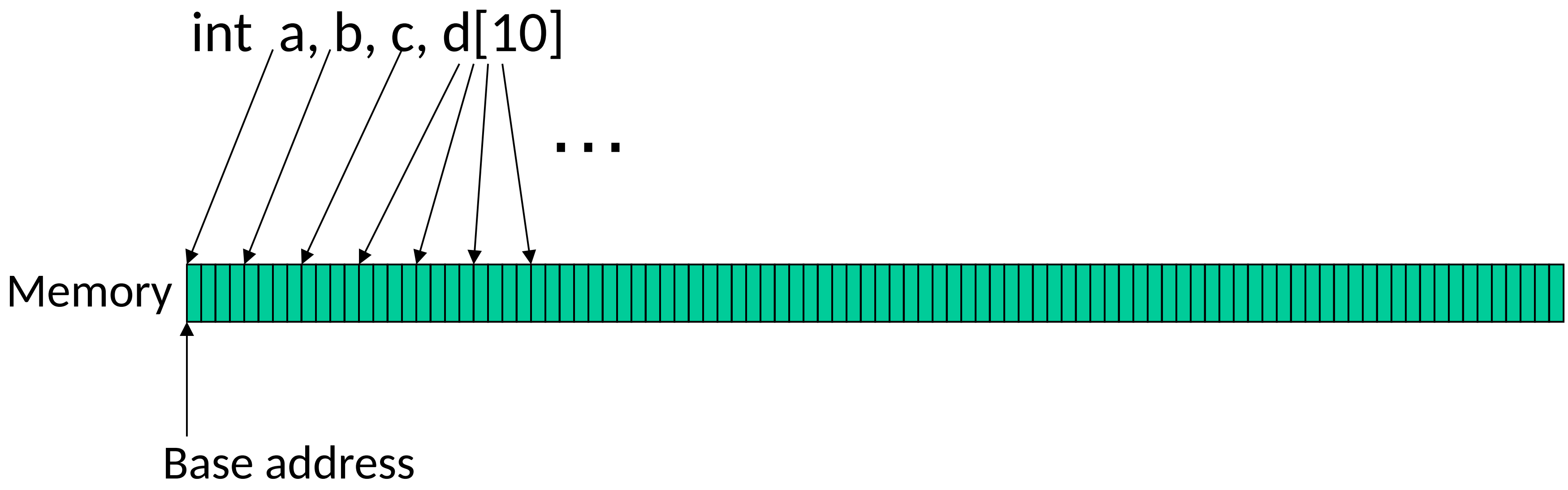
Store word
sw $t0, memory-address



How is memory-address determined?

Memory Address

- The compiler organizes data in memory… it knows the location of every variable (saved in a table)… it can fill in the appropriate mem-address for load-store instructions



int a, b, c, d[10]

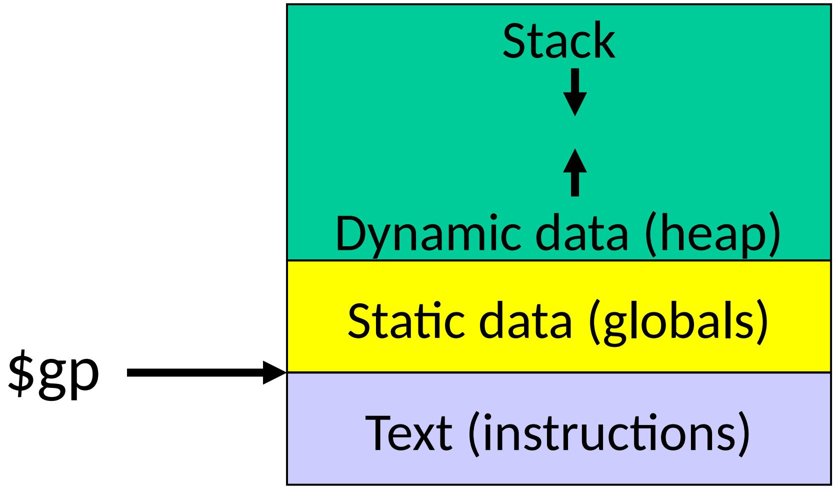
…

Memory

Base address

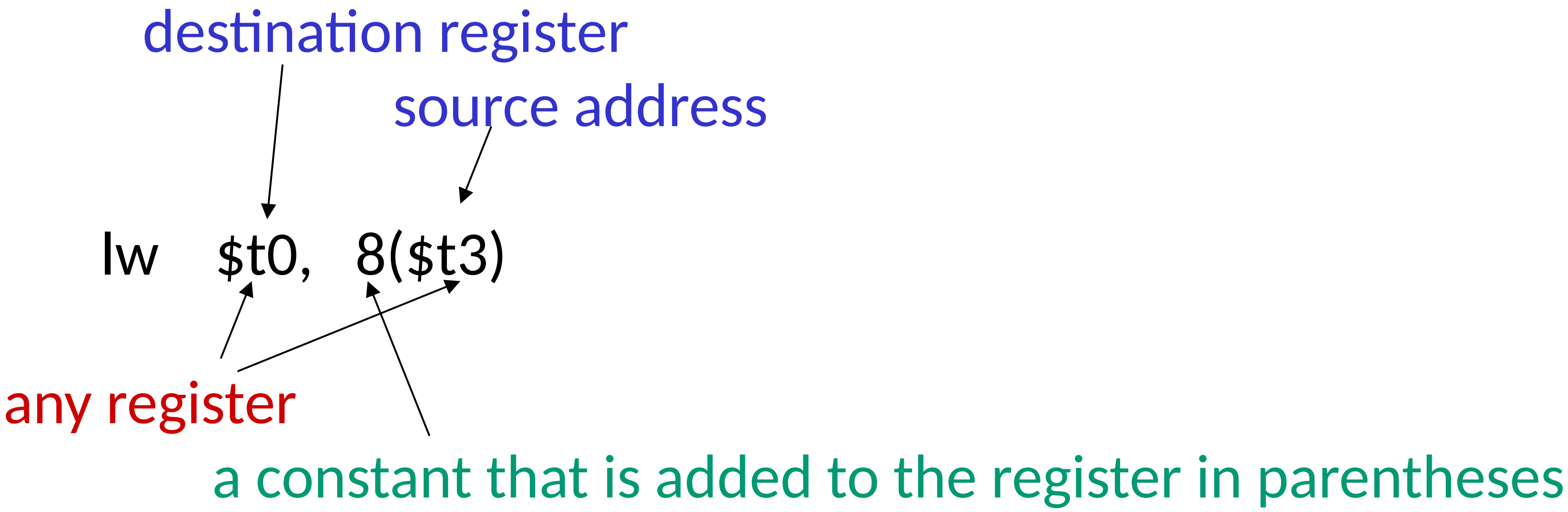
Memory Organization

$gp points to area in memory that saves global variables



Memory Instruction Format

- The format of a load instruction:



destination register

source address

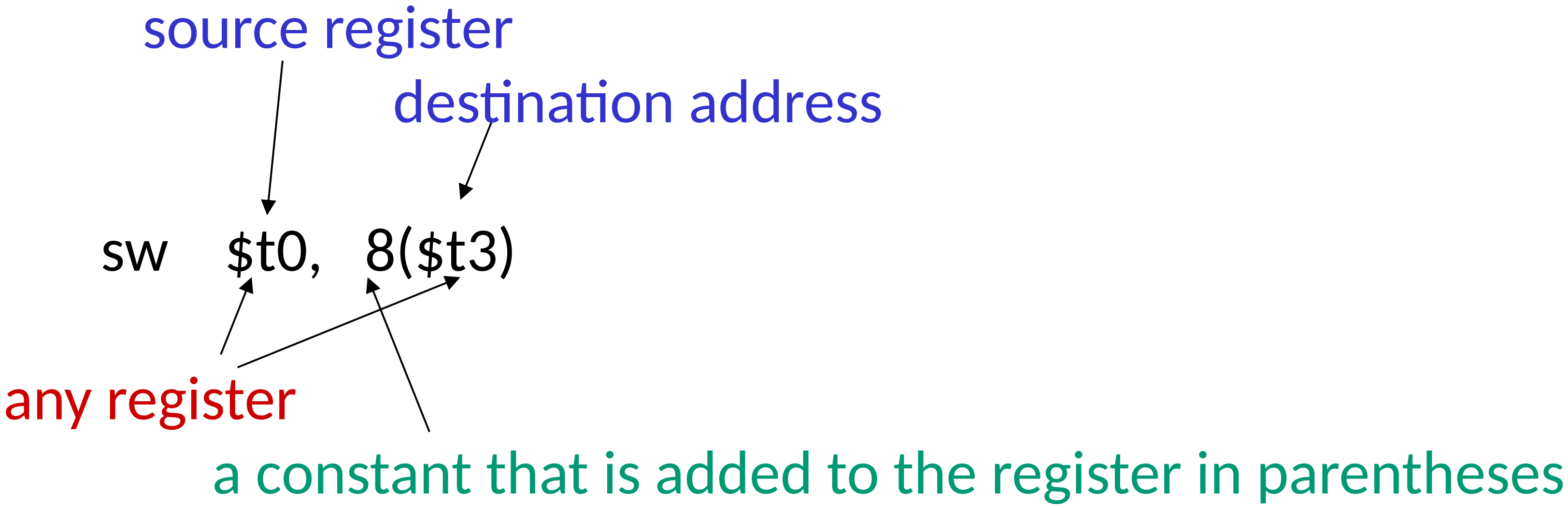
lw $t0, 8($t3)

any register

a constant that is added to the register in parentheses

Memory Instruction Format

- The format of a store instruction:



source register

destination address

sw $t0, 8($t3)

any register

a constant that is added to the register in parentheses

Example

```
int a, b, c, d[10];


addi   $gp, $zero, 1000   # assume that data is stored at
                        # base address 1000; placed in $gp;
                        # $zero is a register that always
                        # equals zero
lw   $s1, 0($gp)         # brings value of a into register $s1
lw   $s2, 4($gp)         # brings value of b into register $s2
lw   $s3, 8($gp)         # brings value of c into register $s3
lw   $s4, 12($gp)        # brings value of d[0] into register $s4
lw   $s5, 16($gp)        # brings value of d[1] into register $s5
```

Example

Convert to assembly:

Remember: int a, b, c, d[10];

C code: d[3] = d[2] + a;

Example

Convert to assembly:
Remember: int a, b, c, d[10];
C code: d[3] = d[2] + a;

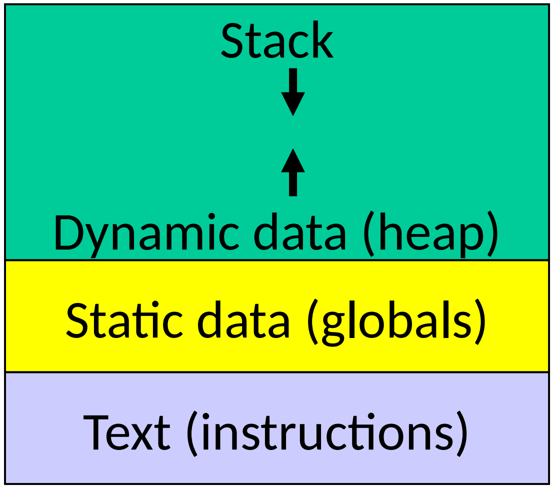
Assembly (same assumptions as previous example):

```
lw      $s0, 0($gp)      #  a is brought into $s0
lw      $s1, 20($gp)     #  d[2] is brought into $s1
add     $s2, $s0, $s1    #  the sum is in $s2
sw      $s2, 24($gp)     #  $s2 is stored into d[3]
```

Assembly version of the code continues to expand!

Memory Organization

- The space allocated on stack by a procedure is termed the activation record (includes saved values and data local to the procedure) – frame pointer points to the start of the record and stack pointer points to the end – variable addresses are specified relative to $fp as $sp may change during the execution of the procedure
- $gp points to area in memory that saves global variables
- Dynamically allocated storage (with malloc()) is placed on the heap



Binary Representation

- The binary number

01011000 00010101 00101110 11100111

Most significant bit

Least significant bit

represents the quantity

$0 \text{ x } 2^{31} + 1 \text{ x } 2^{30} + 0 \text{ x } 2^{29} + \ldots + 1 \text{ x } 2^{0}$

- A 32-bit word can represent 2^{32} numbers between 0 and 2^{32}-1

… this is known as the unsigned representation as we're assuming that numbers are always positive

Negative Numbers

32 bits can only represent 2^{32} numbers – if we wish to also represent negative numbers, we can represent 2^{31} positive numbers (incl zero) and 2^{31} negative numbers

0000 0000 0000 0000 0000 0000 0000 0000$_{two}$ = 0$_{ten}$

0000 0000 0000 0000 0000 0000 0000 0001$_{two}$ = 1$_{ten}$

…

0111 1111 1111 1111 1111 1111 1111 1111$_{two}$ = 2^{31}-1

1000 0000 0000 0000 0000 0000 0000 0000$_{two}$ = -2^{31}

1000 0000 0000 0000 0000 0000 0000 0001$_{two}$ = -(2^{31} – 1)

1000 0000 0000 0000 0000 0000 0000 0010$_{two}$ = -(2^{31} – 2)

…

1111 1111 1111 1111 1111 1111 1111 1110$_{two}$ = -2

1111 1111 1111 1111 1111 1111 1111 1111$_{two}$ = -1

2’s Complement

0000 0000 0000 0000 0000 0000 0000 0000$_{two}$ = 0$_{ten}$

0000 0000 0000 0000 0000 0000 0000 0001$_{two}$ = 1$_{ten}$

…

0111 1111 1111 1111 1111 1111 1111 1111$_{two}$ = 2^{31}-1

1000 0000 0000 0000 0000 0000 0000 0000$_{two}$ = -2^{31}

1000 0000 0000 0000 0000 0000 0000 0001$_{two}$ = -(2^{31} – 1)

1000 0000 0000 0000 0000 0000 0000 0010$_{two}$ = -(2^{31} – 2)

…

1111 1111 1111 1111 1111 1111 1111 1110$_{two}$ = -2

1111 1111 1111 1111 1111 1111 1111 1111$_{two}$ = -1

Why is this representation favorable?

Consider the sum of 1 and -2 …. we get -1

Consider the sum of 2 and -1 …. we get +1

This format can directly undergo addition without any conversions!

Each number represents the quantity

$$x_{31}\ {-2}^{31} + x_{30}\ 2^{30} + x_{29}\ 2^{29} + \ldots + x_1\ 2^1 + x_0\ 2^0$$

2’s Complement

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
…
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}-1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} – 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} – 2)$
…
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

Note that the sum of a number x and its inverted representation x’ always equals a string of 1s (-1).

x + x’ = -1
x’ + 1 = -x ... hence, can compute the negative of a number by
-x = x’ + 1 inverting all bits and adding 1

Similarly, the sum of x and –x gives us all zeroes, with a carry of 1
In reality, x + (-x) = 2^n ... hence the name 2’s complement

Example

- Compute the 32-bit 2’s complement representations for the following decimal numbers:
  5, -5, -6

Example

- Compute the 32-bit 2’s complement representations for the following decimal numbers:
  5, -5, -6

5: 0000 0000 0000 0000 0000 0000 0000 0101
-5: 1111 1111 1111 1111 1111 1111 1111 1011
-6: 1111 1111 1111 1111 1111 1111 1111 1010

Given -5, verify that negating and adding 1 yields the number 5

Recap – Numeric Representations

- Decimal $35_{10} = 3 \text{ x } 10^1 + 5 \text{ x } 10^0$
- Binary $00100011_2 = 1 \text{ x } 2^5 + 1 \text{ x } 2^1 + 1 \text{ x } 2^0$
- Hexadecimal (compact representation)

  $0x\ 23$ or $23_{hex} = 2 \text{ x } 16^1 + 3 \text{ x } 16^0$

0-15 (decimal) ✉ 0-9, a-f (hex)

Dec	Binary	Hex	Dec	Binary	Hex	Dec	Binary	Hex	Dec	Binary	Hex
0	0000	00	4	0100	04	8	1000	08	12	1100	0c
1	0001	01	5	0101	05	9	1001	09	13	1101	0d
2	0010	02	6	0110	06	10	1010	0a	14	1110	0e
3	0011	03	7	0111	07	11	1011	0b	15	1111	0f

Constant or Immediate Operands

- We often use constants in operations
- Example: add 4 to register $s3

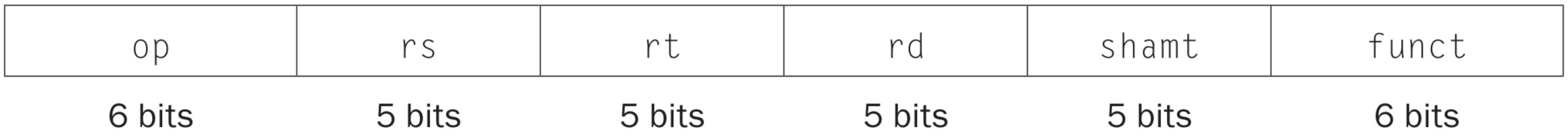
```
lw $t0, AddrConstant4($s1)# $t0 = constant 4
add $s3,$s3,$t0 # $s3 = $s3 + $t0 ($t0 == 4)
```

- A more elegant way

```
addi $s3,$s3,4 # $s3 = $s3 + 4
```

Instruction format (R-type)

- Instructions are 32bit words in memory

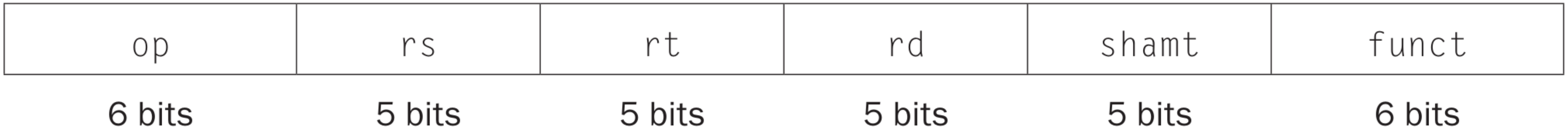


op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- *op*: Basic operation of the instruction, traditionally called the opcode
- *rs*: The first register source operand
- *rt*: The second register source operand
- *rd*: The register destination operand. It gets the result of the operation
- *shamt*: Shift amount
- *funct*: Function/function code, selects the specific variant of the operation in the op field

Instruction format (R-type)

- Instructions are 32bit words in memory

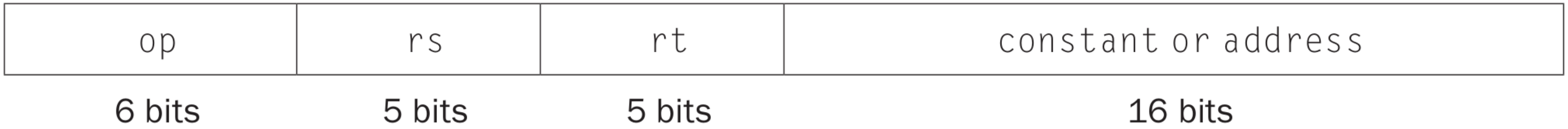


op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- This works ok well for instructions like
  - add $s0, $s1, $s3
- But what about
  - `lw $t0, 32($s0)`
  - `addi $t0, $t1, 4  # t0 = t1 + 4`

Instruction format (I-type)

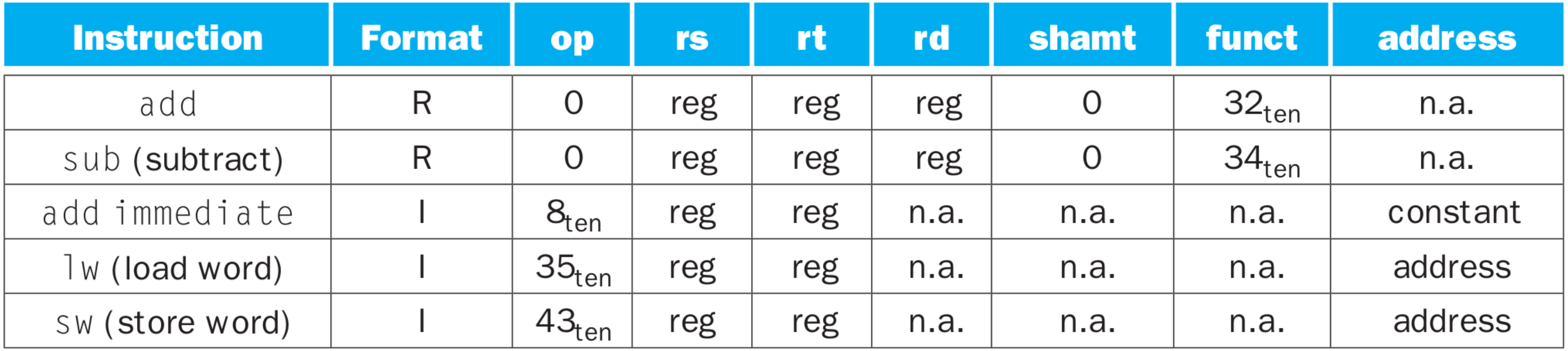
- Instructions are 32bit words in memory



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

- *op*: Basic operation of the instruction, traditionally called the opcode
- *rs*: The first register source operand
- *rt*: New meaning – destination register

Examples



Instruction	Format	op	rs	rt	rd	shamt	funct	address
`add`	R	0	reg	reg	reg	0	32_{ten}	n.a.
`sub` (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
`add immediate`	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
`lw` (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
`sw` (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

FIGURE 2.5 MIPS instruction encoding. In the table above, “reg” means a register number between 0 and 31, “address” means a 16-bit address, and “n.a.” (not applicable) means this field does not appear in this format. Note that add and sub instructions have the same value in the op field; the hardware uses the funct field to decide the variant of the operation: add (32) or subtract (34).

Register numbers

- $s0 - $s7 map on hardware registers 16 – 23
  - E.g., $s0 is 16, $s1 is 17
- $t0 - $t7 map on hardware registers 8 – 15
  - E.g., $t0 is 8, $t1 is 17

Example

```
A[300] = h + A[300]
```

- Gets compiled to

Example

```
A[300] = h + A[300]
```

- Gets compiled to

```
lw  $t0,1200($t1) # Temporary reg $t0 gets A[300]

add $t0,$s2,$t0   # Temporary reg $t0 gets h + A[300]

sw  $t0,1200($t1) # Stores h+A[300] back into A[300]
```

Example

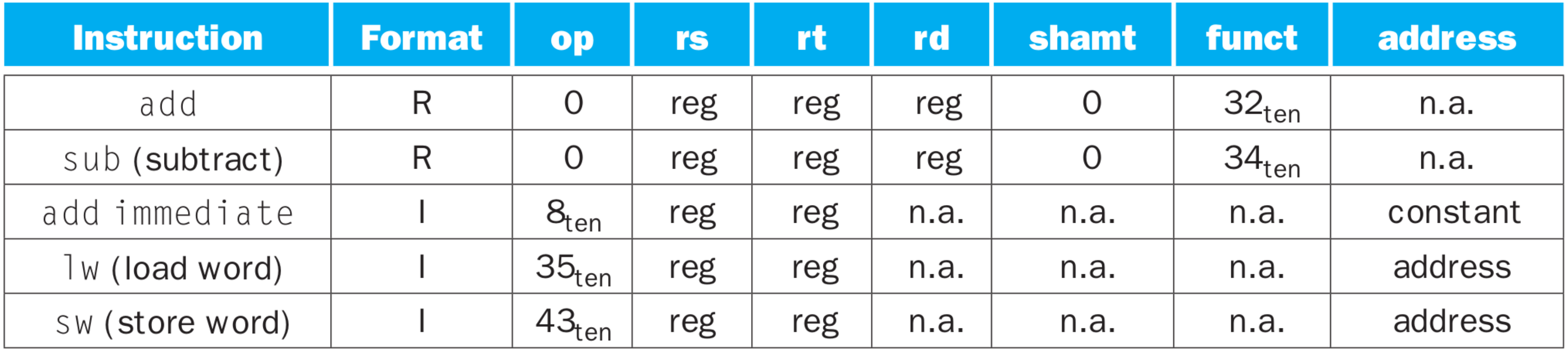
```
A[300] = h + A[300]
```

- Gets compiled to

```
lw  $t0,1200($t1) # Temporary reg $t0 gets A[300]
add $t0,$s2,$t0   # Temporary reg $t0 gets h + A[300]
sw  $t0,1200($t1) # Stores h+A[300] back into A[300]
```

<table>
<tr><th>Op</th><th>rs</th><th>rt</th><th>rd</th><th>address/
shamt</th><th>funct</th></tr>
<tr><td>35</td><td>9</td><td>8</td><td colspan="3">1200</td></tr>
<tr><td>0</td><td>18</td><td>8</td><td>8</td><td>0</td><td>32</td></tr>
<tr><td>43</td><td>9</td><td>8</td><td colspan="3">1200</td></tr>
</table>

Instruction encoding



Instruction	Format	op	rs	rt	rd	shamt	funct	address
`add`	R	0	reg	reg	reg	0	32_{ten}	n.a.
`sub` (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
`add immediate`	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
`lw` (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
`sw` (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

FIGURE 2.5 MIPS instruction encoding. In the table above, "reg" means a register number between 0 and 31, "address" means a 16-bit address, and "n.a." (not applicable) means this field does not appear in this format. Note that add and sub instructions have the same value in the op field; the hardware uses the funct field to decide the variant of the operation: add (32) or subtract (34).

Example

- Decimal

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

- Binary

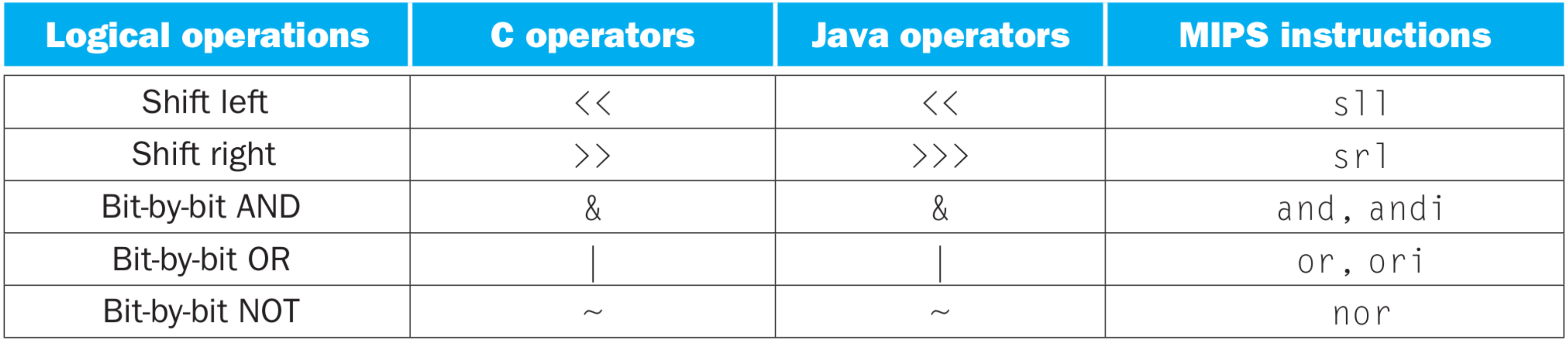
100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add `$s1,$s2,$s3`
sub	R	0	18	19	17	0	34	sub `$s1,$s2,$s3`
addi	I	8	18	17	100			addi `$s1,$s2,100`
lw	I	35	18	17	100			lw `$s1,100($s2)`
sw	I	43	18	17	100			sw `$s1,100($s2)`
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

FIGURE 2.6 MIPS architecture revealed through Section 2.5. The two MIPS instruction formats so far are R and I. The first 16 bits are the same: both contain an *op* field, giving the base operation; an *rs* field, giving one of the sources; and the *rt* field, which specifies the other source operand, except for load word, where it specifies the destination register. R-format divides the last 16 bits into an *rd* field, specifying the destination register; the *shamt* field, which Section 2.6 explains; and the *funct* field, which specifies the specific operation of R-format instructions. I-format combines the last 16 bits into a single *address* field.

Logical operations



Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bit-by-bit AND	&	&	`and, andi`
Bit-by-bit OR	|	|	`or, ori`
Bit-by-bit NOT	~	~	`nor`

FIGURE 2.8 C and Java logical operators and their corresponding MIPS instructions. MIPS implements NOT using a NOR with one operand being zero.

Shift

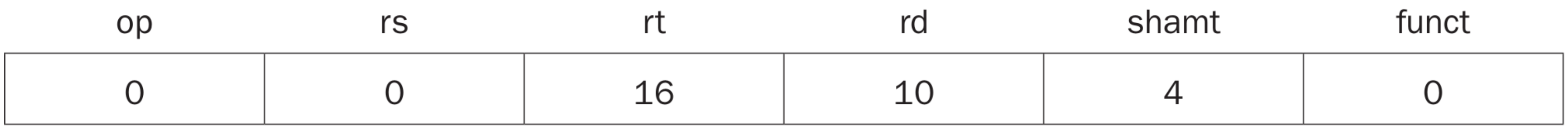
- Shift left by 4
- Before
  - 0000 0000 0000 0000 0000 0000 0000 1001 = 9
- After
  - 0000 0000 0000 0000 0000 0000 1001 0000 = 144

Encoding shift

- Example

```
sll $t2,$s0,4 # reg $t2 = reg $s0 << 4 bits
```

- Shift amount



op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

Control instructions

Branch instructions

- Branch when equal

  ```
  beq register1, register2, L1
  ```

  - Go to L1 if `register1` equals `register2`
- Branch when not equal

  ```
  bne register1, register2, L1
  ```

  - Go to L1 if `register1` does *not* equal `register2`
- Unconditional jump
  - `j L1`
  - Jump to L1

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:
```

Comparisons

- Set on less than

```
slt $t0, $s3, $s4 # $t0 = 1 if $s3 < $s4
```

- Or with a constrant

```
slti $t0,$s2,10 # $t0 = 1 if $s2 < 10
```

- Now you can use slt, slti, beq, and bne along with $zero (register that is always 0)

Branch instructions

- Branch when equal

  ```
  beq register1, register2, L1
  ```

  - Go to L1 if `register1` equals `register2`
- Branch when not equal

  ```
  bne register1, register2, L1
  ```

  - Go to L1 if `register1` does *not* equal `register2`
- Unconditional jump
  - `j L1`
  - Jump to L1

Branch instructions

- Branch when equal

  ```
  beq register1, register2, L1
  ```

  - Go to L1 if `register1` equals `register2`
- Branch when not equal

  ```
  bne register1, register2, L1
  ```

  - Go to L1 if `register1` does *not* equal `register2`
- Unconditional jump
  - `j L1`
  - Jump to L1

Shift Operations

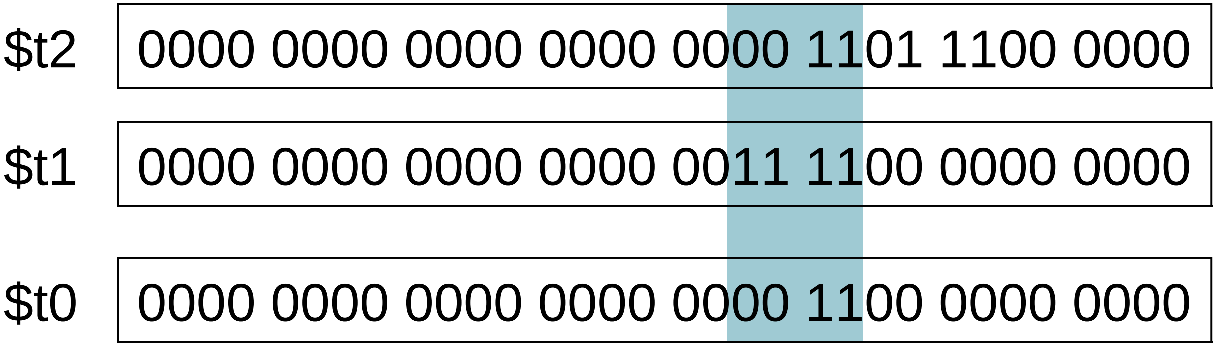
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and $t0, $t1, $t2
```

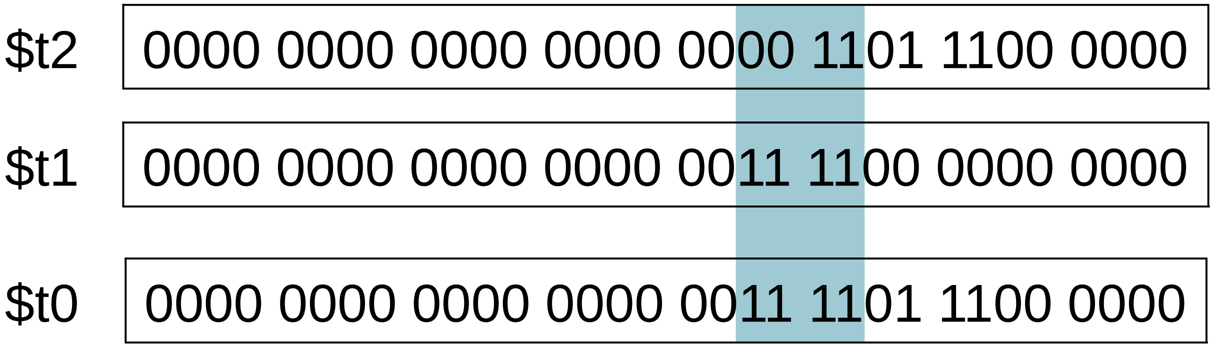


$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```



$t2 0000 0000 0000 0000 0000 1101 1100 0000

$t1 0000 0000 0000 0000 0011 1100 0000 0000

$t0 0000 0000 0000 0000 0011 1101 1100 0000

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1 0000 0000 0000 0000 0011 1100 0000 0000

$t0 1111 1111 1111 1111 1100 0011 1111 1111

Signed and unsigned comparisons

Consider a comparison instruction:

```
    slt   $t0, $t1, $zero
```

and $t1 contains the 32-bit number   1111 01…01

What gets stored in $t0?

Signed and unsigned comparisons

Consider a comparison instruction:

```
    slt   $t0, $t1, $zero
```

and $t1 contains the 32-bit number   1111 01…01

What gets stored in $t0?

The result depends on whether $t1 is a signed or unsigned number – the compiler/programmer must track this and accordingly use either slt or sltu

```
  slt    $t0, $t1, $zero     stores  1 in $t0
  sltu  $t0, $t1, $zero     stores  0 in $t0
```

Sign Extension

- Occasionally, 16-bit signed numbers must be converted into 32-bit signed numbers – for example, when doing an add with an immediate operand

- The conversion is simple: take the most significant bit and use it to fill up the additional bits on the left – known as sign extension

So 2_{10} goes from 0000 0000 0000 0010 to
0000 0000 0000 0000 0000 0000 0000 0010

and -2_{10} goes from 1111 1111 1111 1110 to
1111 1111 1111 1111 1111 1111 1111 1110

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
```

Save $s0 on stack

Procedure body

Result

Restore $s0

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
```

Local variables

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)

```
#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

Dynamic variables (heap)

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;
}
```

Local variables...

- Each function has private instances of local variables

```
foo(int x) {
    int a, b, c;
    ...
    return;
}
```

- Function can be called recursively

```
foo(int x) {
    int a, b, c;
    a = x + 1;
    if ( a < 100 )
        foo(a);
    return;
}
```

How to allocate local variables?

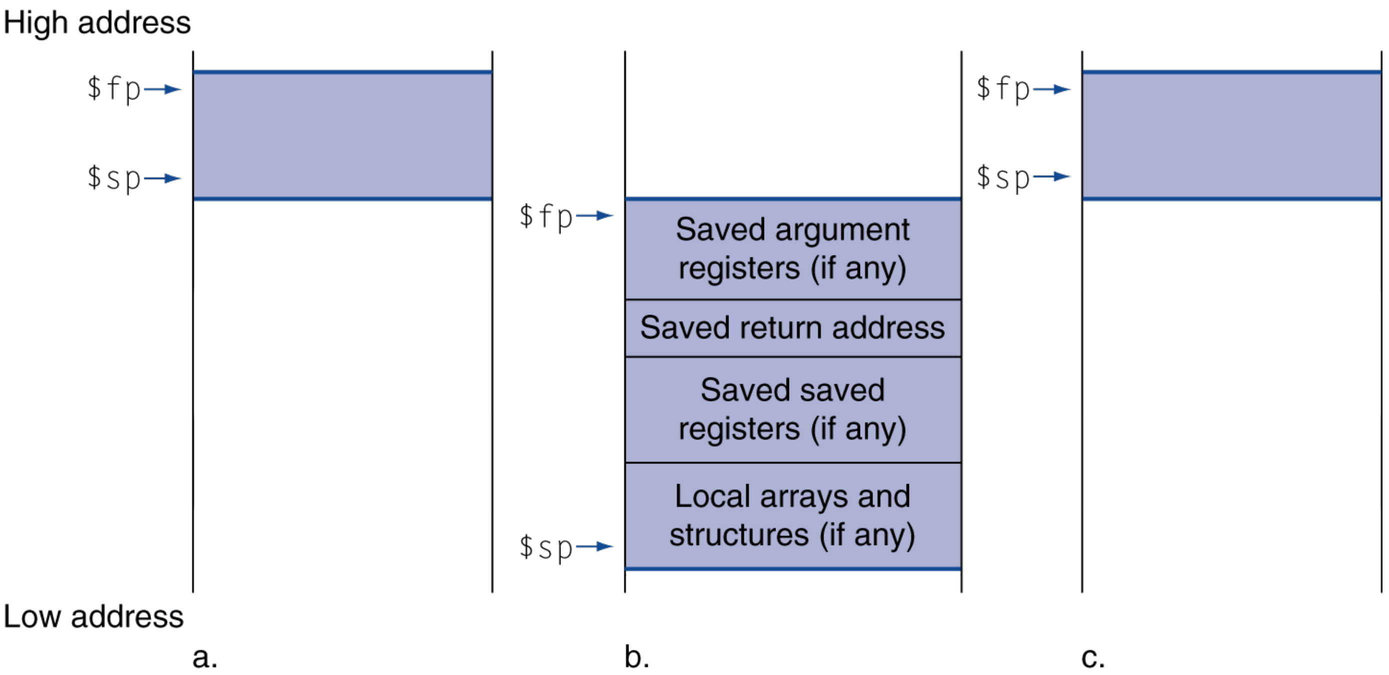
```
void my_function()
{
    int a, b, c;
    …
}
```

How to allocate local variables?

```
void my_function()
{
    int a, b, c;
    …
}
```

- On the stack!

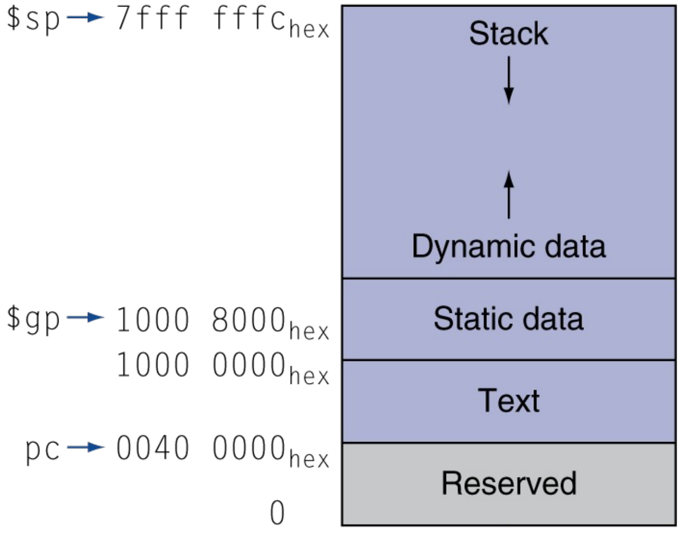
Local Data on the Stack


High address
$fp
$sp
$fp
$sp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$fp
$sp
Low address
a.
b.
c.

- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Memory Layout

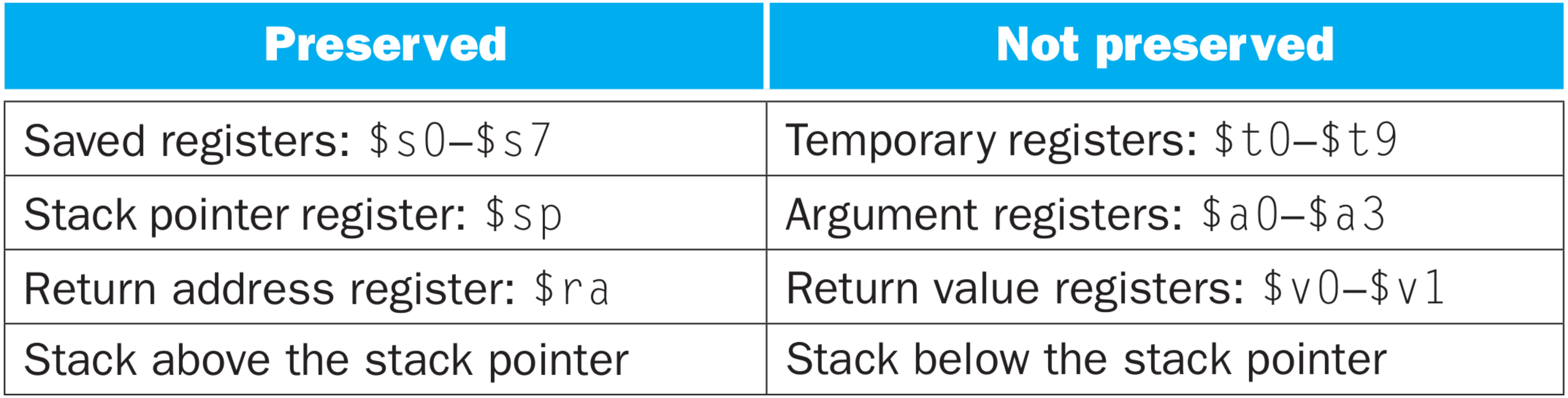
- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Recap: Procedure Calling

- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure’s operations
  5. Place result in register for caller
  6. Return to place of call

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

Binary numbers, << 1, mul 2

Strings

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world’s alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

§2.9 Communicating with People

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

```
lb rt, offset(rs)     lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)    lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)     sh rt, offset(rs)
```

  - Store just rightmost byte/halfword

String Copy Example

- C code (naïve):
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Addresses of x, y in $a0, $a1
  - i in $s0

String Copy Example

- 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
```

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

```
lui rt, constant
```

  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

`lhi $s0, 61`	0000 0000 0111 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101	0000 1001 0000 0000

§2.10 MIPS Addressing for 32-Bit Immediates and Addresses

J-Type instructions

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

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

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits

- (Pseudo)Direct jump addressing
  - Target address = $PC_{31…28}$: (address × 4)

Branching Far Away

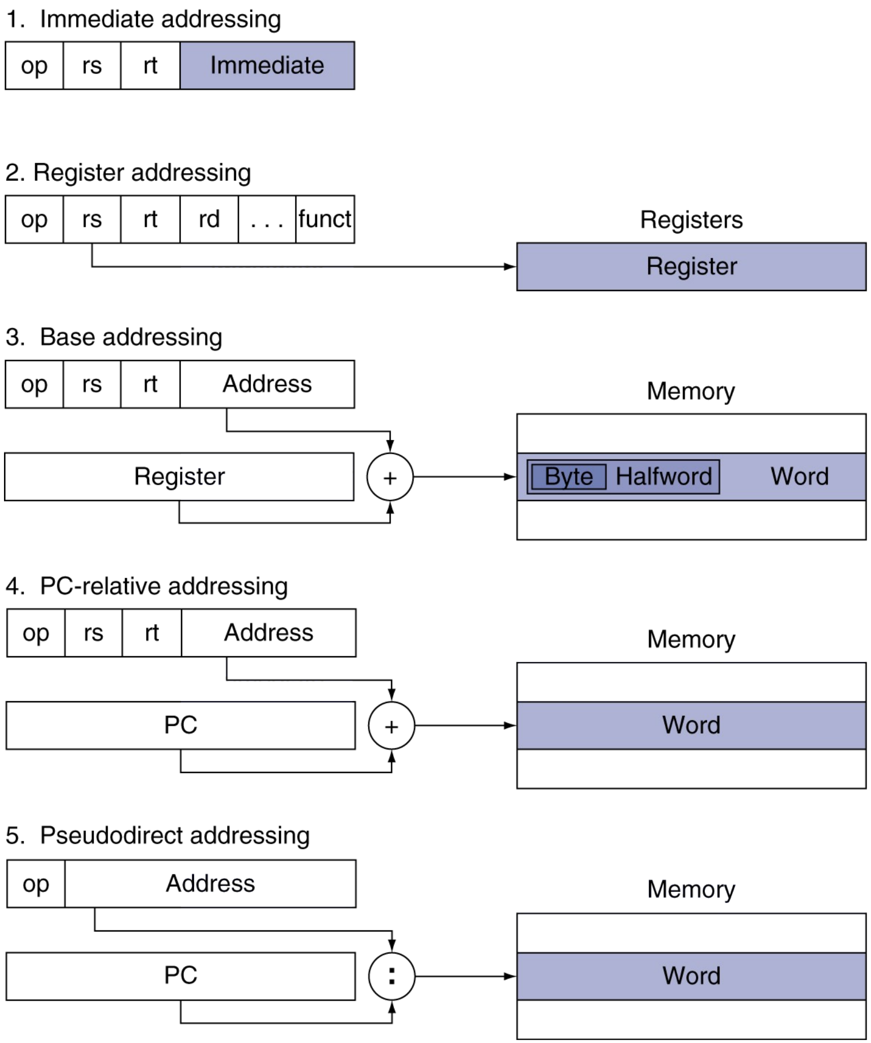
- If branch target is too far to encode with 16-bit offset, assembler rewrites the code

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
    beq $s0,$s1, L1
          ↓
    bne $s0,$s1, L2
    j L1
L2: …
```

Addressing Mode Summary



MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

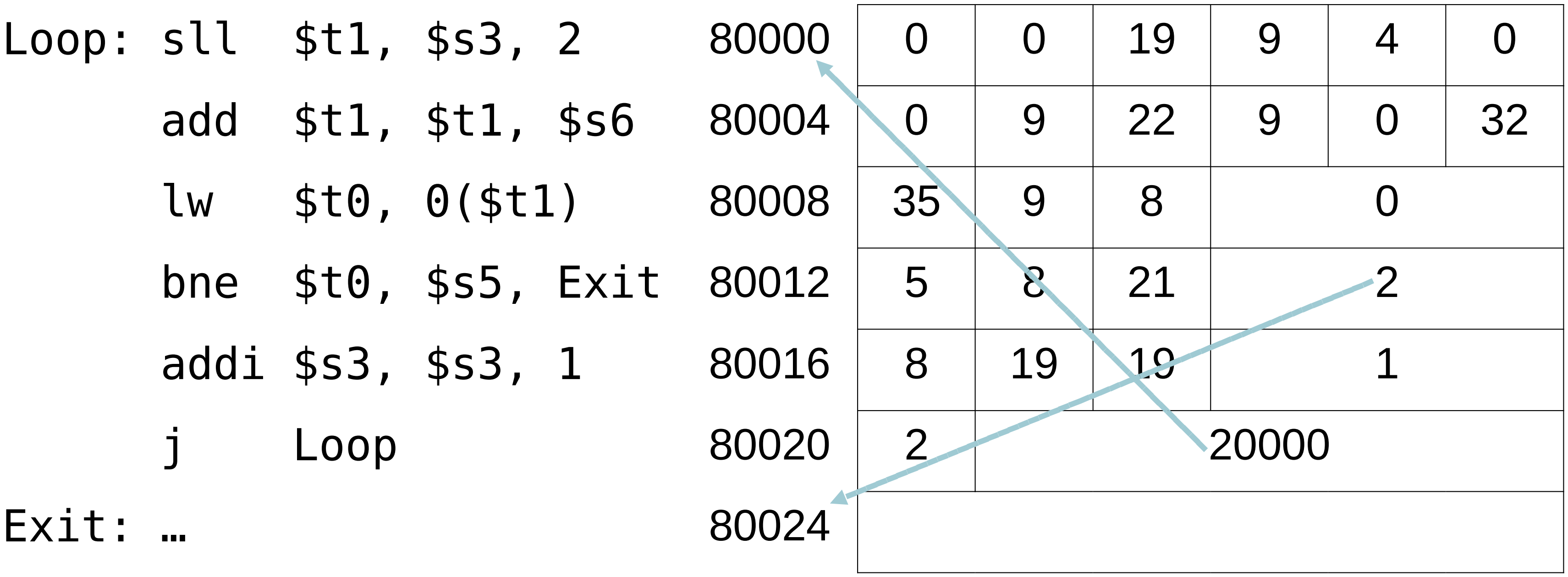
MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits

Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

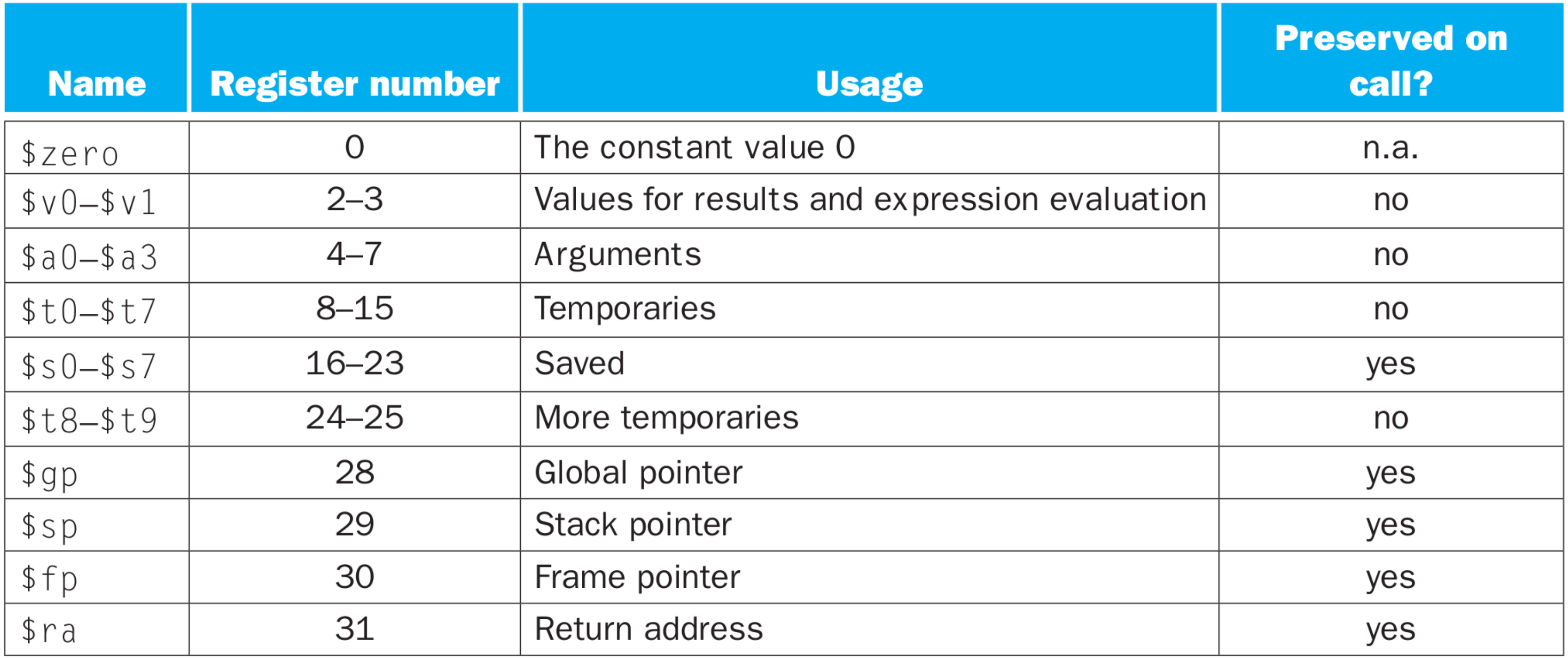
Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000



`Loop: sll  $t1, $s3, 2`	80000	0	0	19	9	4	0
`      add  $t1, $t1, $s6`	80004	0	9	22	9	0	32
`      lw   $t0, 0($t1)`	80008	35	9	8	0		
`      bne  $t0, $s5, Exit`	80012	5	8	21	2		
`      addi $s3, $s3, 1`	80016	8	19	19	1		
`      j    Loop`	80020	2	20000				
`Exit: …`	80024						

Recap: registers



Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Linking and loading

What is inside a program?

- What parts do we need to run code?

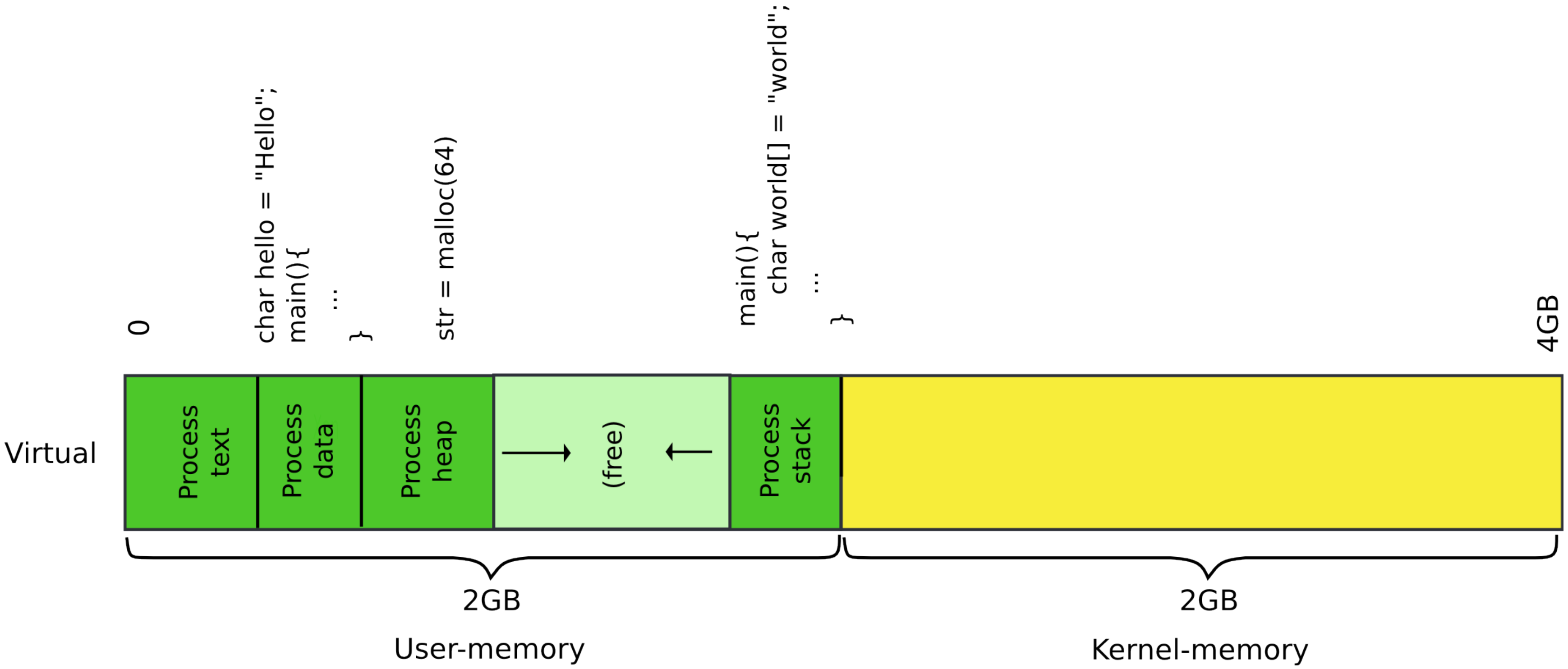
Parts needed to run a program

- Code itself
  - By convention it's called text
- Stack
  - To call functions
- Space for variables

What types of variables do you know?

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

Memory layout of a process

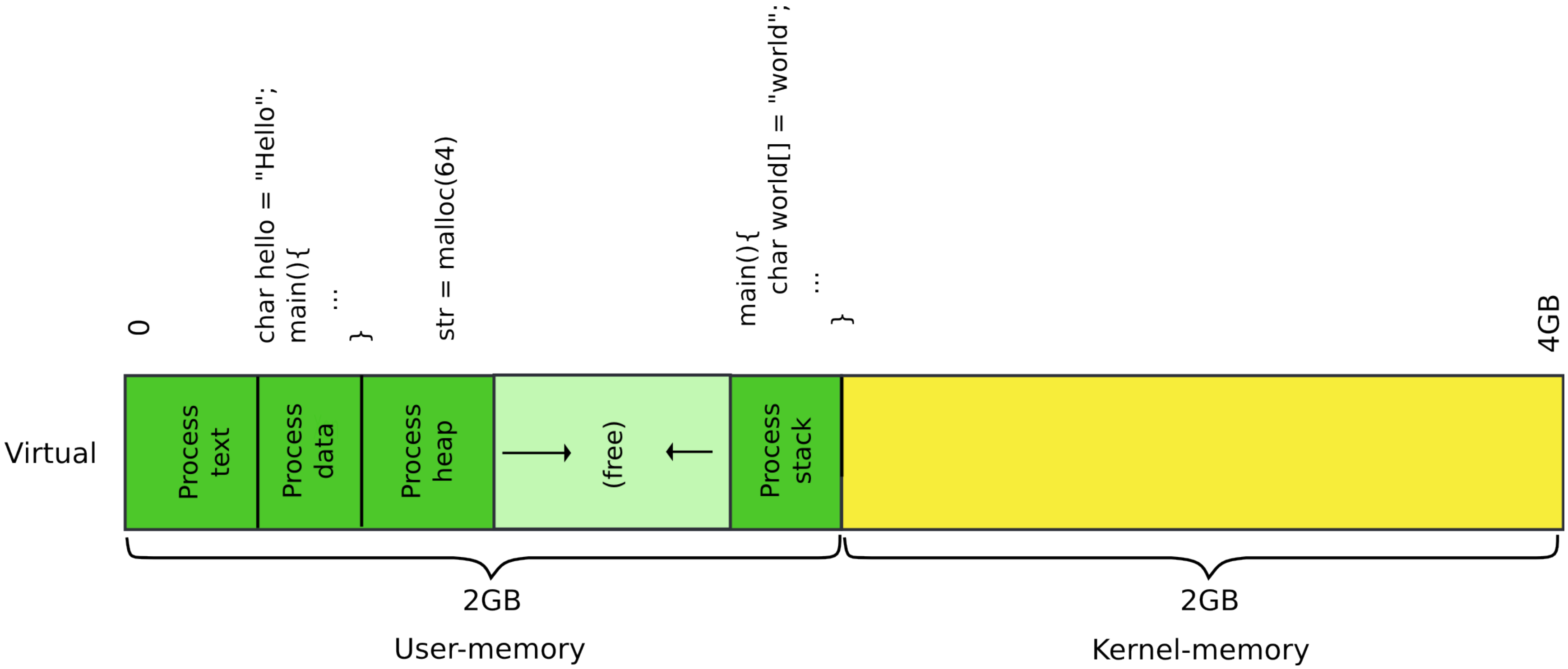


```
char hello = "Hello";
main(){
    ...
}
```

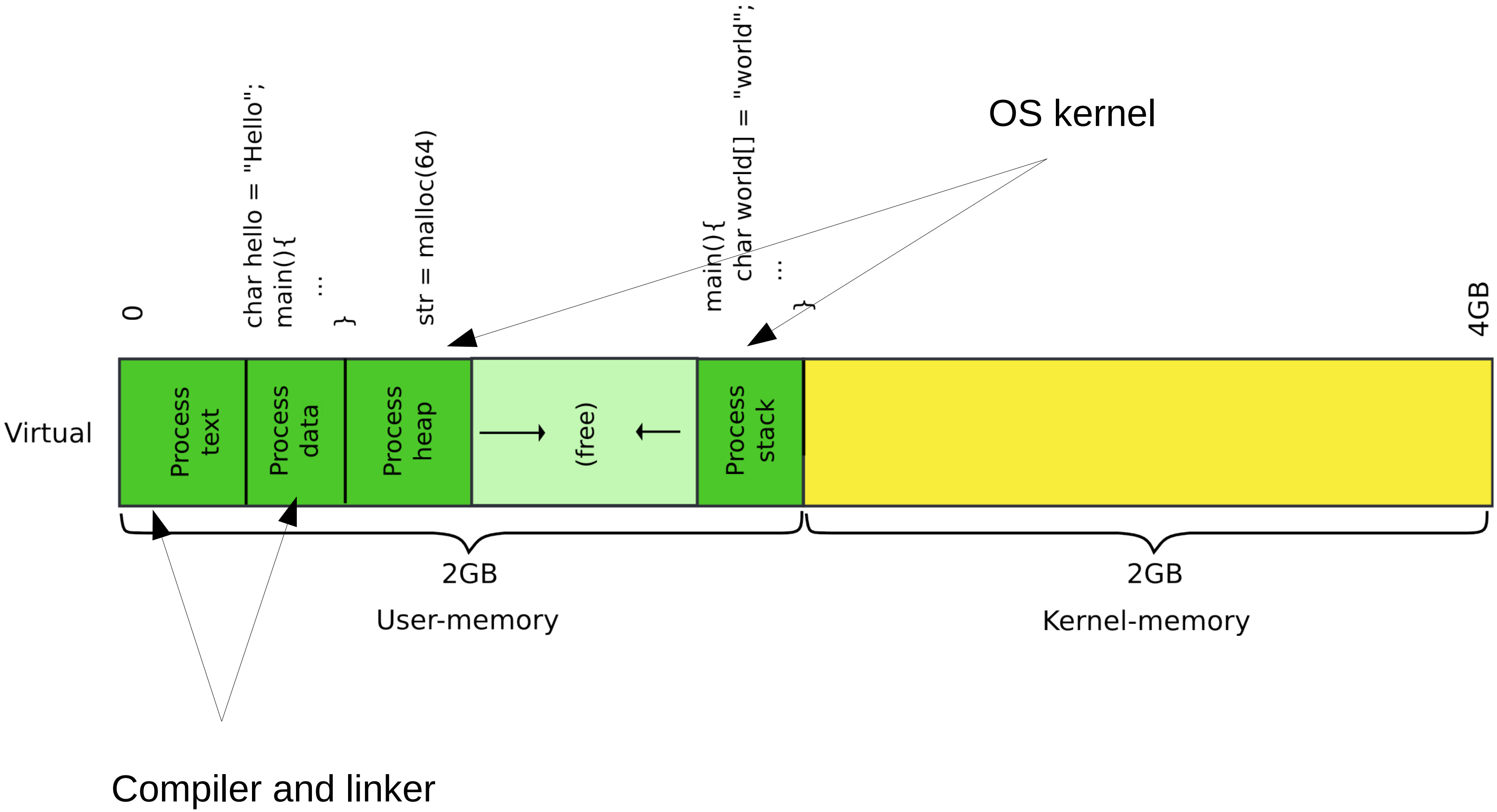
```
str = malloc(64)
```

```
main(){
  char world[] = "world";
    ...
}
```

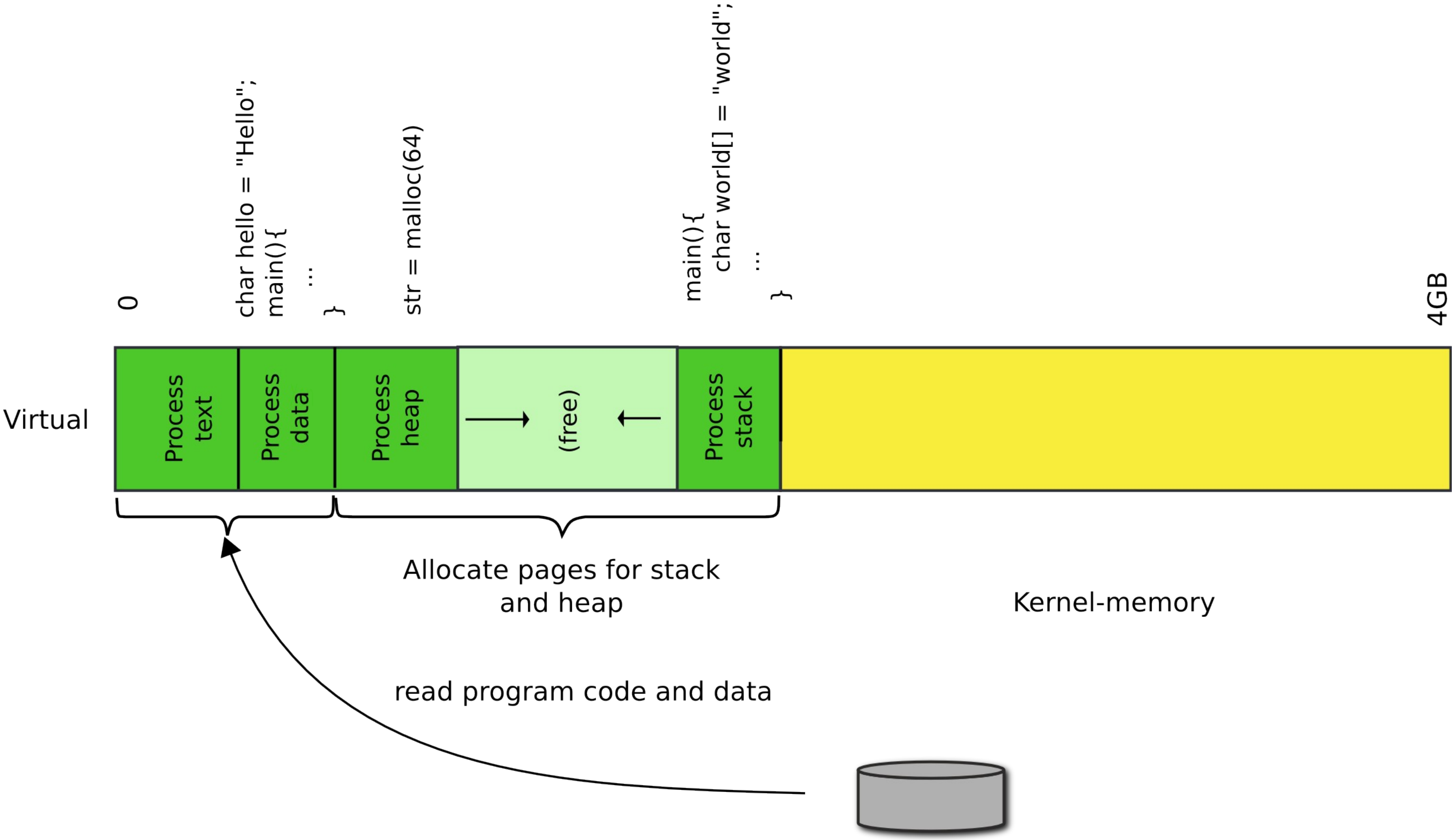
Where do these areas come from?



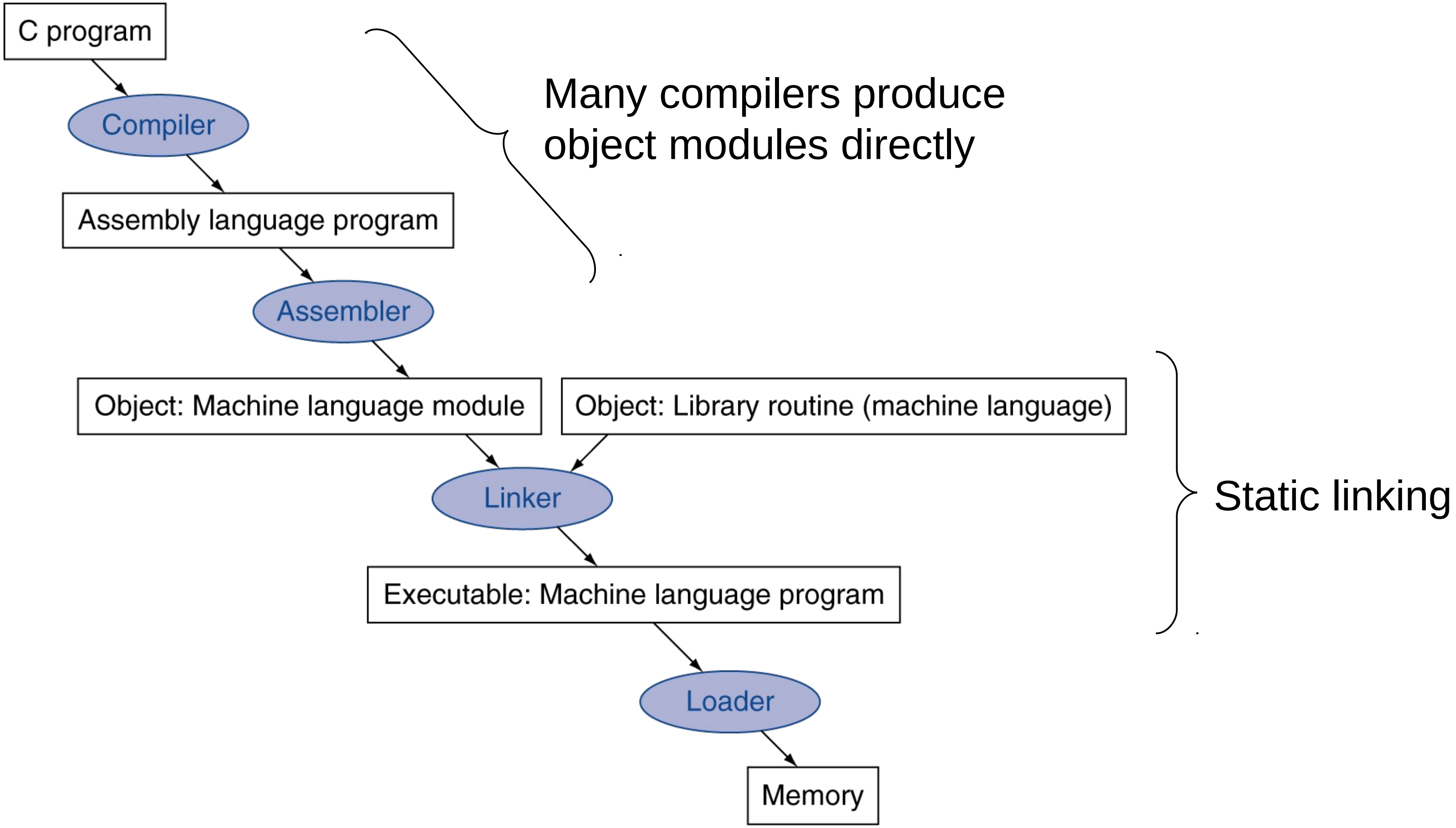
Memory layout of a process



Load program in memory



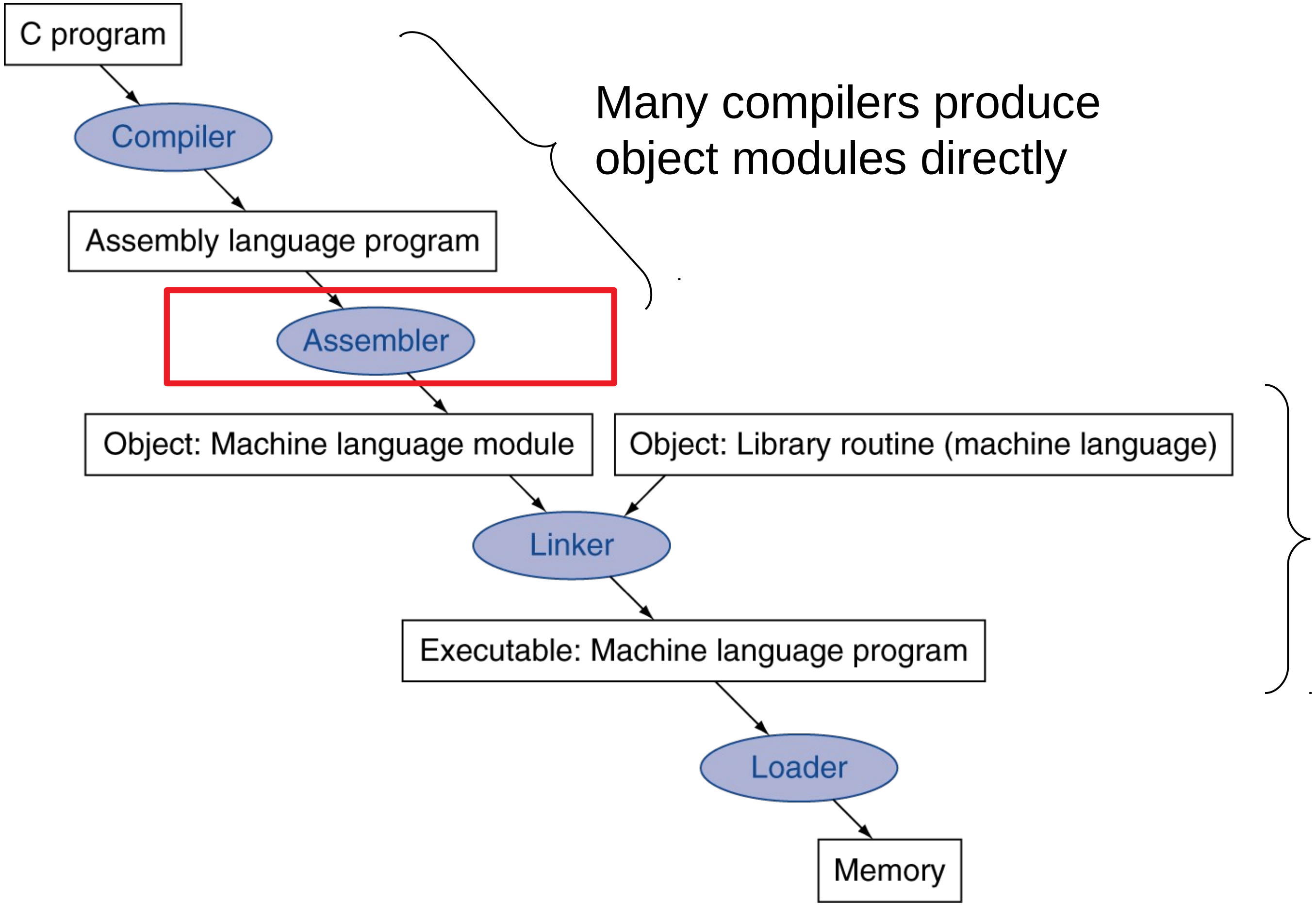
Translation and Startup


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

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

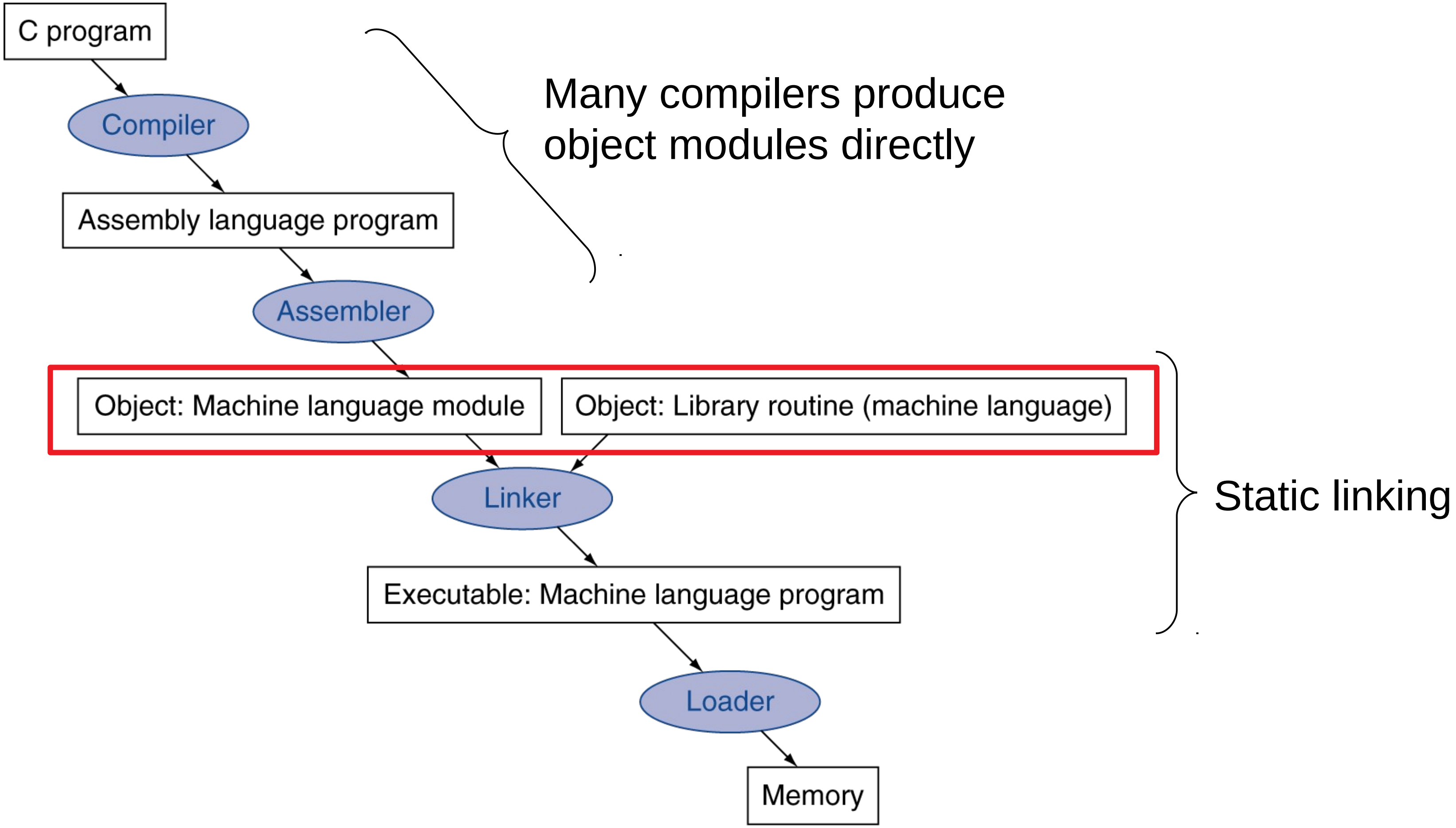
Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: high-level assembly constructs

```
move $t0, $t1       →  add $t0, $zero, $t1
blt $t0, $t1, L      →  slt $at, $t0, $t1
                        bne $at, $zero, L
```

  - $at (register 1): assembler temporary

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

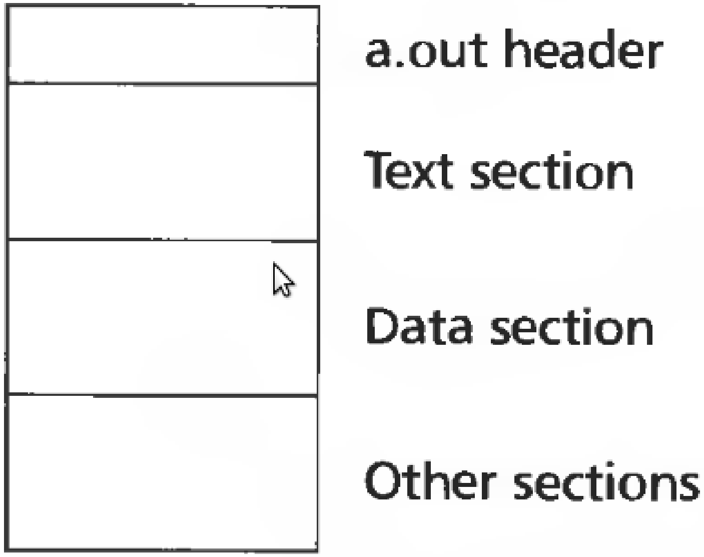
Static linking

Object files (.o)

Object files

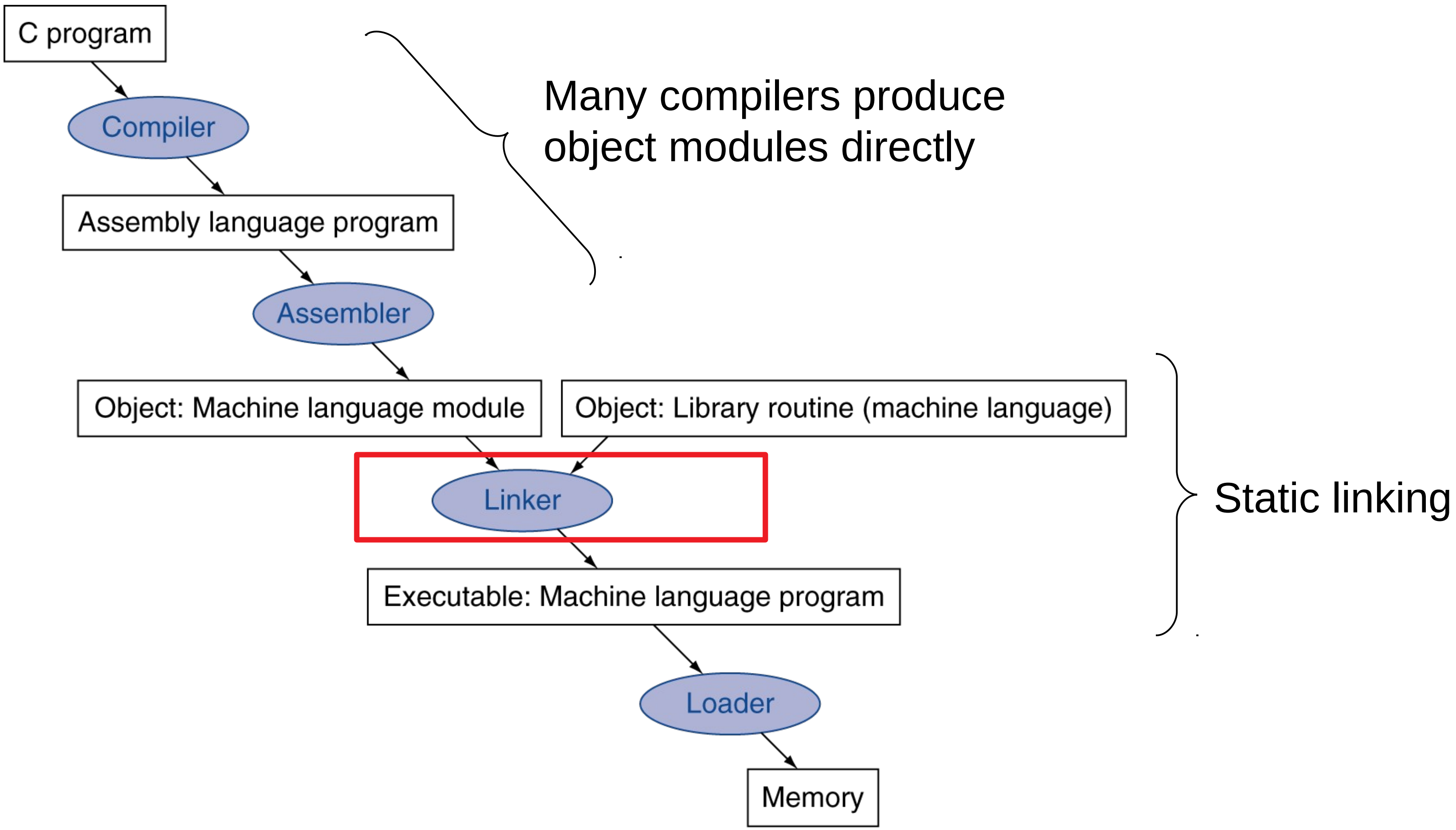
- Conceptually: five kinds of information
  - Header: code size, name of the source file, creation date
  - Object code: binary instruction and data generated by the compiler
  - Relocation information: list of places in the object code that need to be patched
  - Symbols: global symbols defined by this module
    - Symbols to be imported from other modules
  - Debugging information: source file and file number information, local symbols, data structure description

Example: UNIX A.OUT



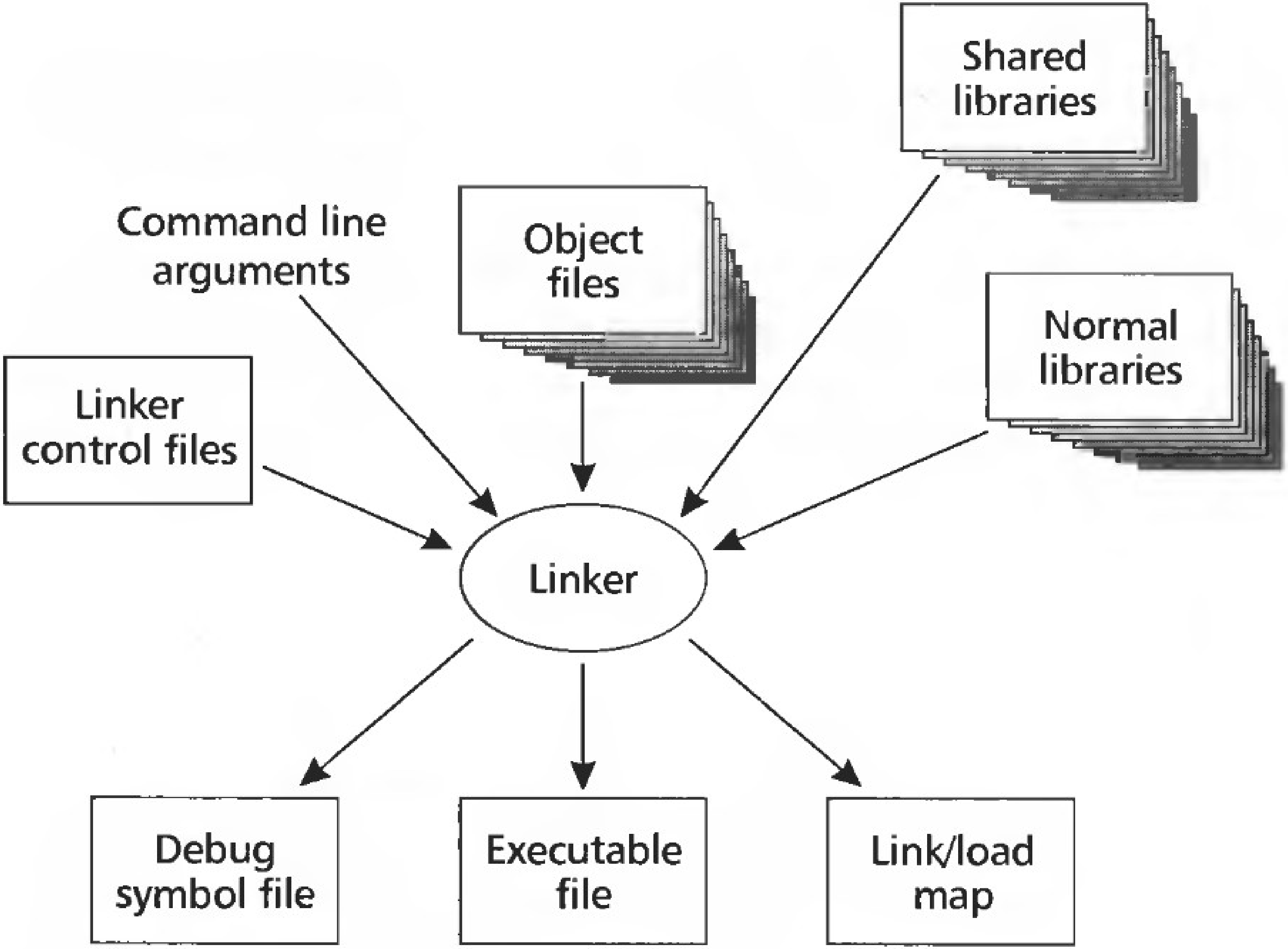
- Small header
- Text section
  - Executable code
- Data section
  - Initial values for static data

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

Why linking?

Why linking?

- Modularity
  - Program can be written as a collection of modules
  - We can build libraries of common functions
- Efficiency
  - Code compilation
    - Change one source file, recompile it, and re-link the executable
  - Space efficiency
    - Share common code across executable files
    - On disk and in memory

Example: printf()

- 1000 programs in a typical UNIX system
- 1000 copies of printf

- **How big is `printf()` actually?**

Motivation

- Disk space
  - 2504 programs in /usr/bin on my Linux laptop
    - `ls /usr/bin | wc -l`
  - **`printf()` is a large function**
  - **Handles conversion of multiple types to strings**
    - **5-10K**
  - This means 10-25MB of disk can be wasted just on `printf()`
- Runtime memory costs are
  - 5-10K times the number of running programs
  - 250 programs running on my Linux laptop
    - `ps -aux | wc -l`
    - 1MB-2.5MB – huge number for most systems 15-20 years ago

Two kinds of linking

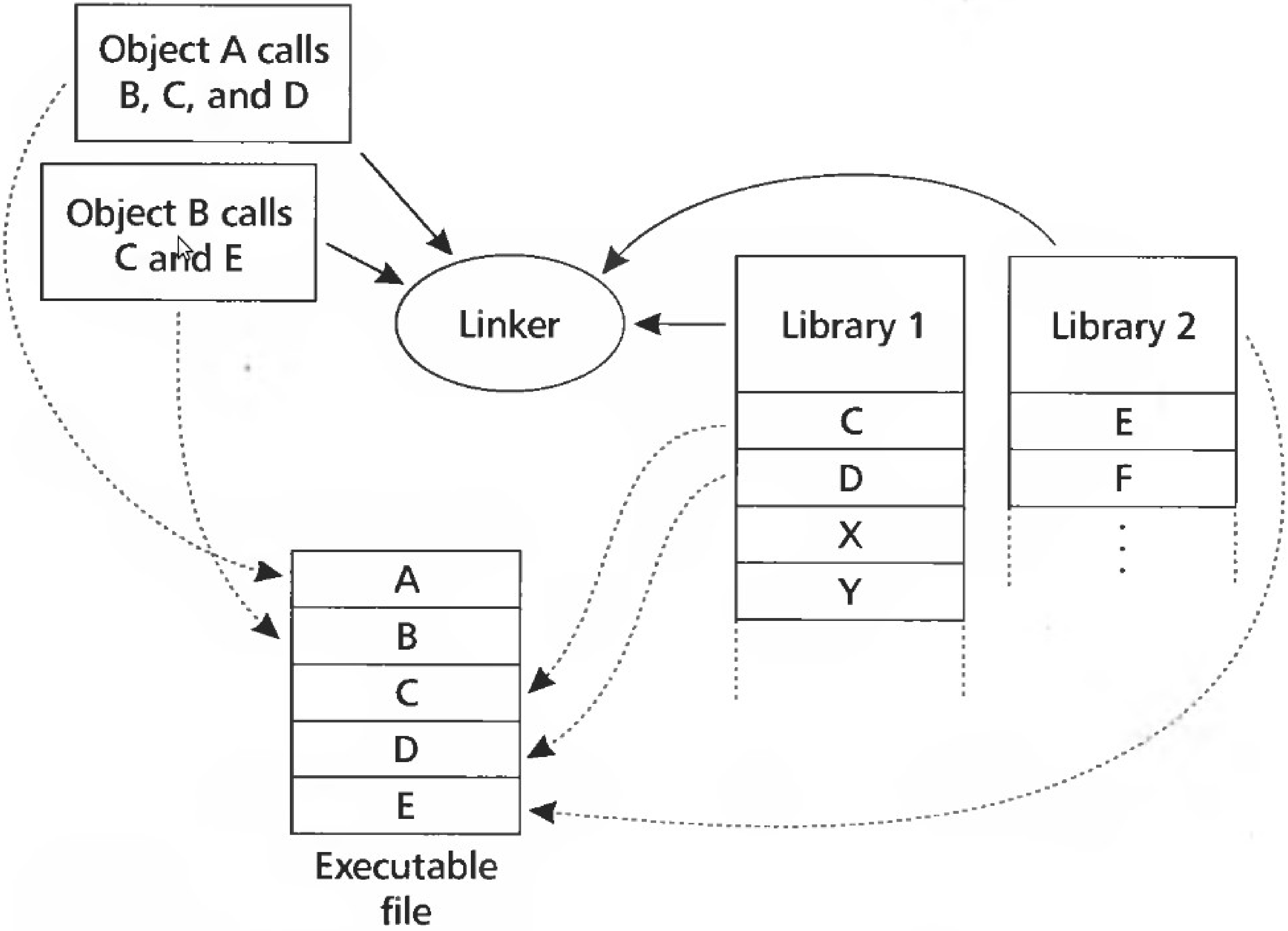
- Static
  - The program is linked at compilation time
  - `main() + static libraries => executable`
- Dynamic
  - The program is linked right when it’s loaded into memory
  - `main() + dynamic libraries => executable`

Example: size of a statically vs dynamically linked program

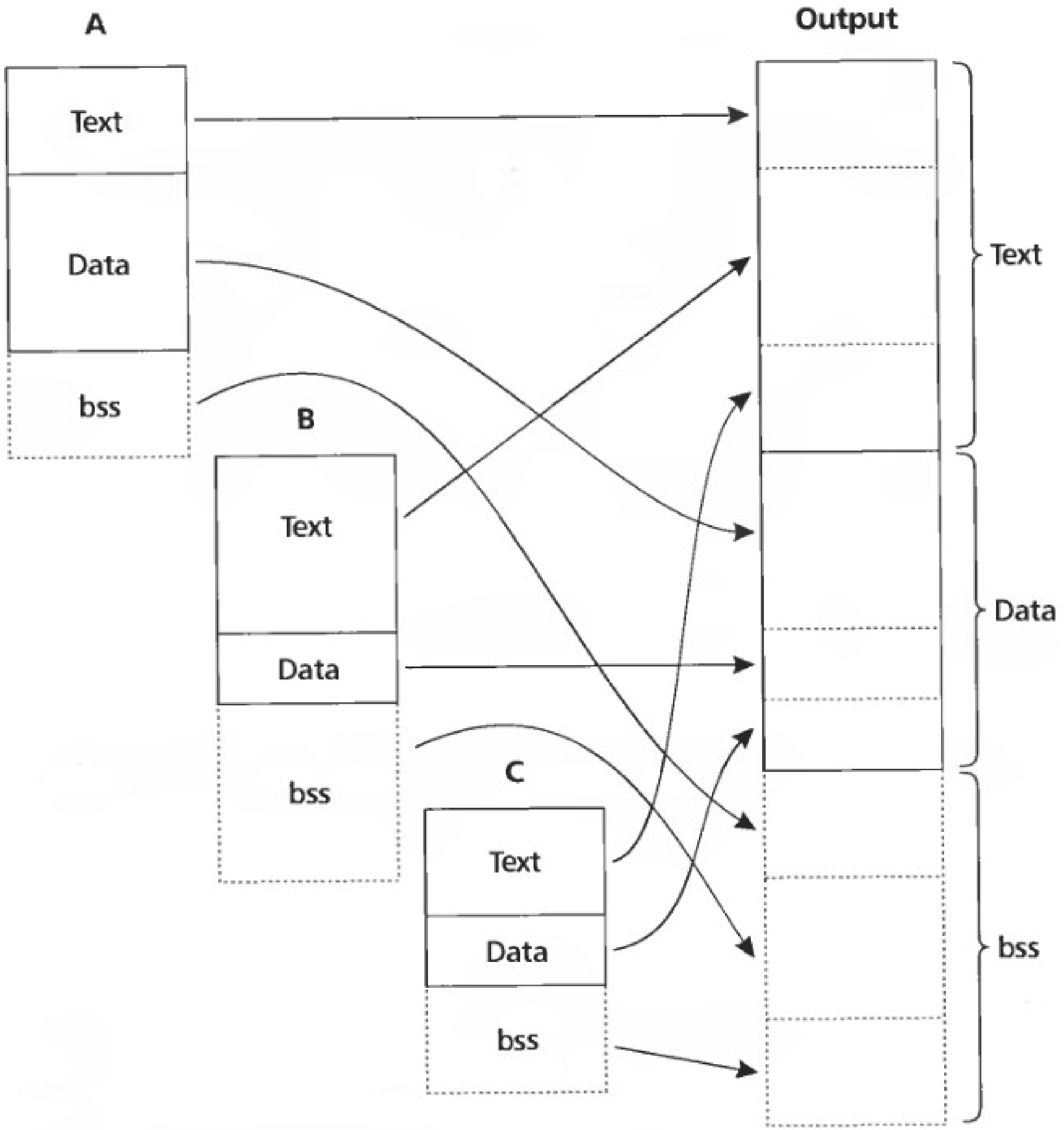
- On Ubuntu 16.04 (gcc 5.4.0, libc 2.23)
  - Statically linked trivial example
    - `gcc -m32 -static hello-int.c -o test`
    - 725KB
  - Dyncamically linked trivial example
    - `gcc -m32 hello-int.c -o test`
    - 7KB

Linking

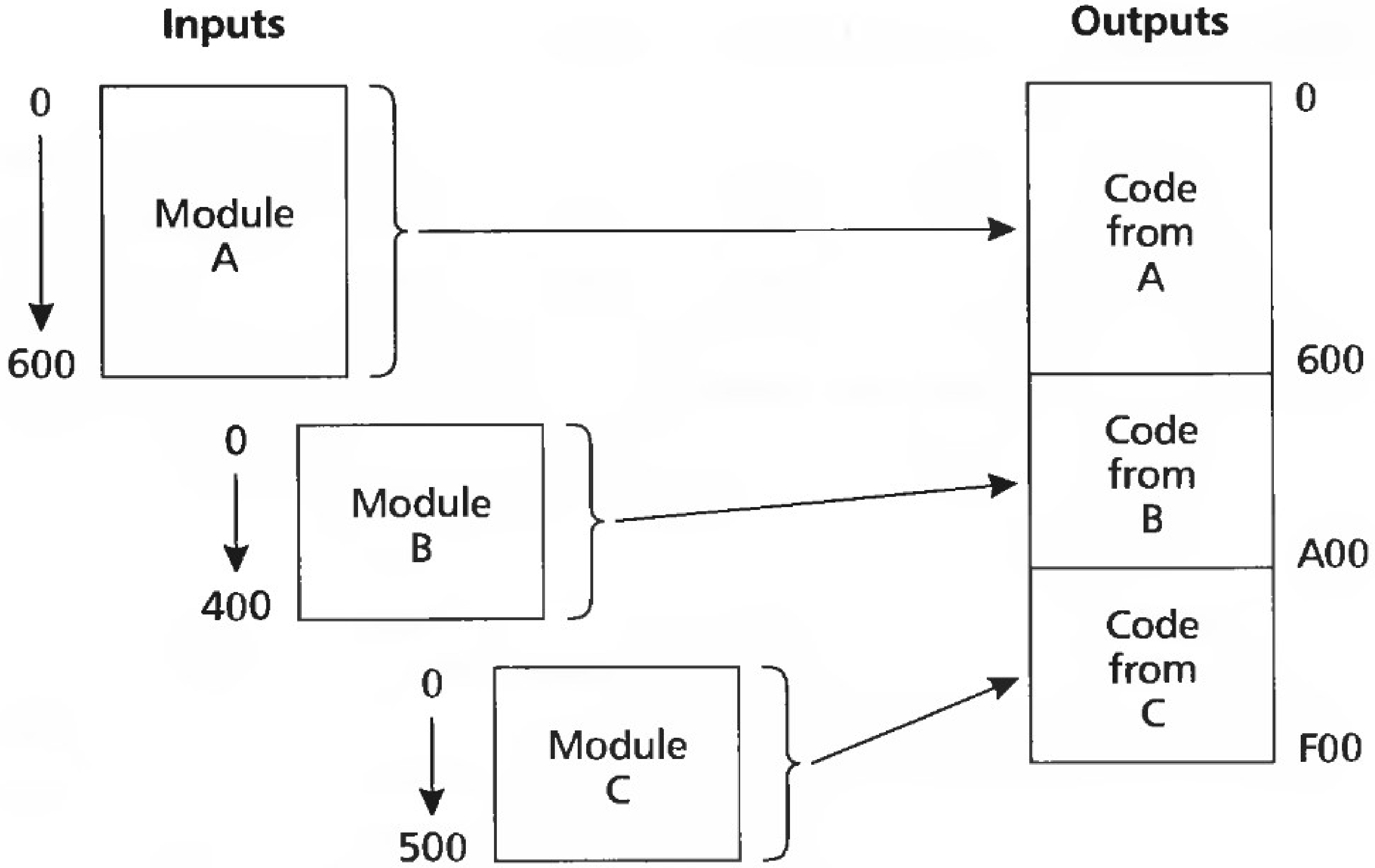
- Input: object files (code modules)
- Each object file contains
  - A set of segments
    - Code
    - Data
  - A symbol table
    - Imported & exported symbols
- Output: executable file, library, etc.


Object A calls B, C, and D
Object B calls C and E
Linker
Library 1
C
D
X
Y
Library 2
E
F
A
B
C
D
E
Executable file

Merging segment s



Merging code

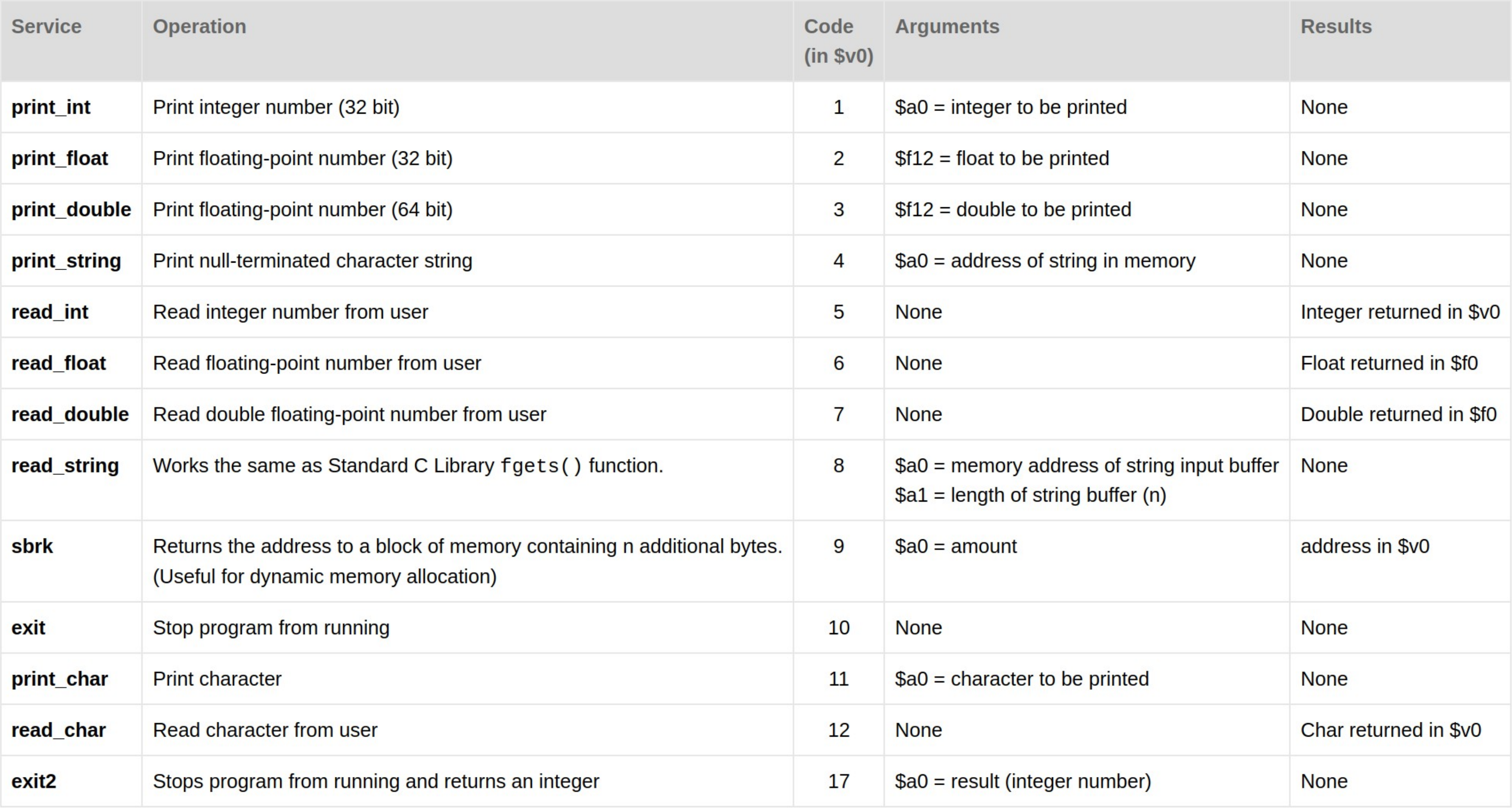


What needs to be done to merge (or move) code in memory?

Detour: real programs

```
# 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:

...

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

...
```

System calls



Service	Operation	Code (in $v0)	Arguments	Results
print_int	Print integer number (32 bit)	1	$a0 = integer to be printed	None
print_float	Print floating-point number (32 bit)	2	$f12 = float to be printed	None
print_double	Print floating-point number (64 bit)	3	$f12 = double to be printed	None
print_string	Print null-terminated character string	4	$a0 = address of string in memory	None
read_int	Read integer number from user	5	None	Integer returned in $v0
read_float	Read floating-point number from user	6	None	Float returned in $f0
read_double	Read double floating-point number from user	7	None	Double returned in $f0
read_string	Works the same as Standard C Library `fgets()` function.	8	$a0 = memory address of string input buffer $a1 = length of string buffer (n)	None
sbrk	Returns the address to a block of memory containing n additional bytes. (Useful for dynamic memory allocation)	9	$a0 = amount	address in $v0
exit	Stop program from running	10	None	None
print_char	Print character	11	$a0 = character to be printed	None
read_char	Read character from user	12	None	Char returned in $v0
exit2	Stops program from running and returns an integer	17	$a0 = result (integer number)	None

```
# "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

...
```

```
# "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"
```

Main’s signature

```
int main(int argc, char* argv[], char **envp) {
   while (*envp != NULL) {
     printf("%s\n", *envp++);
   }
   return 0;
 }
```

```
$ gcc t.c
$ ./a.out
SHELL=/bin/bash
TERM=xterm-256color
HISTSIZE=1000
EDITOR=vim
LANG=en_US.UTF-8
HISTCONTROL=ignoredups
ARCH=x86_64
DISPLAY=:0
COLORTERM=truecolor
...
```

Relocation

```
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
```

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main
     - When main returns, do exit syscall

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

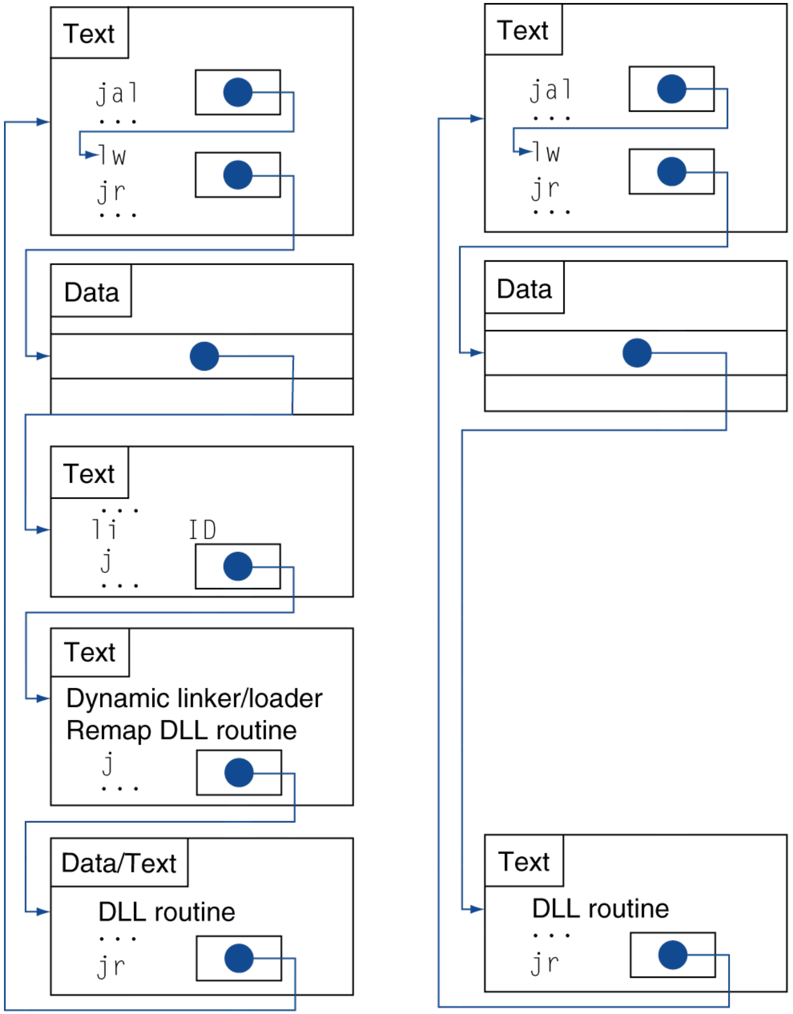
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine