Computer Technology

Topic 1

Basic structure of a computer. Assembly language programming.

[Adapted from the book *Computer Organization and Design, 4th Edition*,Patterson & Hennessy, © 2008, MK]

Overview

- Introduction
  - What is a computer?
  - Programming a processor
  - Running a program
- Instruction set architecture
  - Instruction set
  - Instruction set, Pat-Hen 2.1-2.3
  - Instruction formats Pat-Hen 2.5
- Assembly language programming 2.5-2.8, 2.10
  - Introduction to MIPS processor
  - MIPS assembly language
  - Translation of high-level structures to MIPS assembly language

What is a computer?

INTRODUCTION

What is a computer?

- A **computer** is a general purpose device (system) with two elements
  - **Hardware (HW)**: Circuits and wires
  - **Software (SW)**: a collection of programs and data that provides the information for telling HW what to do and how
  - What is the difference between a **calculator** and a **computer**?

What is a computer?

- A **computer** is a general purpose device (system) with two components
  - **HW**: Circuits and wires
  - **SW**: a collection of programs and data that provides the information for telling HW what to do and how to do it

How?

Controlling HW $\Rightarrow$ Indicating what to do

What is a computer?

- A **computer** is a general purpose device (system) with two components
  - **HW**: Circuits and wires
  - **SW**: a collection of programs and data that provides the information for telling HW what to do and how to do it

How?

Controlling HW $\Rightarrow$ Indicating what to do

How?

Sequence of instructions specifying what HW has to do in order to perform a specified task designed to solve a problem. What kind of problems?

What is a computer?

- A **computer** is a general purpose device (system) with two components
  - **HW**: Circuits and wires
  - **SW**: a collection of programs and data that provides the information for telling HW what to do and how to do it

How?

Sequence of instructions that specifies what HW has to do in order to perform a specified task to solve a problem

PROGRAM

Classes of Computers

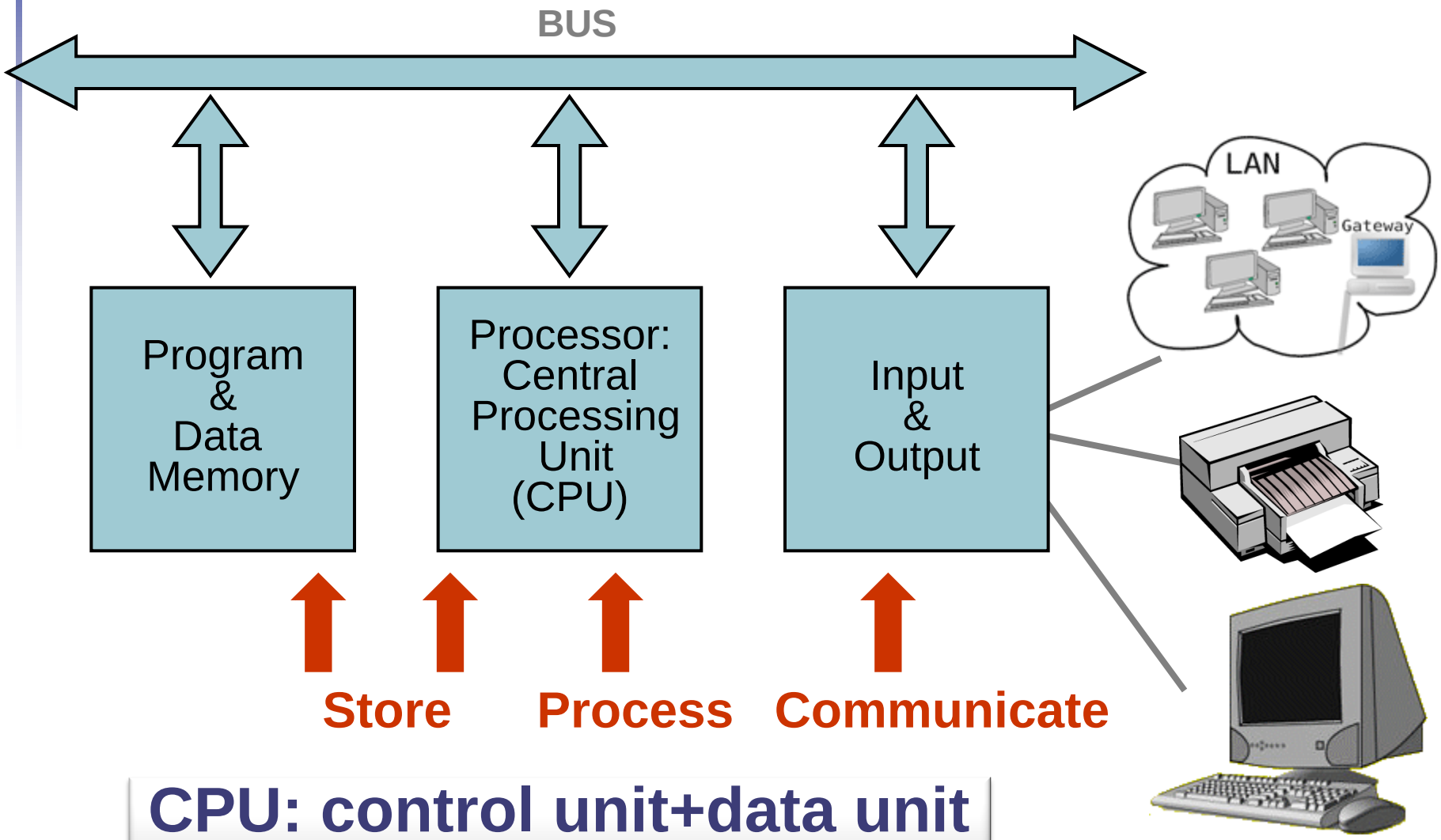
Depending on its purpose, the design can be different

- Special-purpose computers
  - Designed to perform specific tasks
  - Embedded computers (processors)
    - A computer inside another device used for running one predetermined application, e.g. board computer
    - Cell phones, tablets, smart phones, TV, washing machines, cars,….
- General- purpose computers
  - Designed to perform general tasks
  - Desktop computers, servers, supercomputers, …

Classes of Computers

- Special-purpose computers
  - Designed to perform specific tasks
  - Embedded computers (processors)
    - A computer inside another device used for running one predetermined application
    - Cell phones, tablets, smart phones, TV, washing machines, cars,….
- General- purpose computers
  - Designed to perform general tasks
  - Desktop computers, servers, supercomputers, …
- **Stored-program concept:** John Von Neumann (1945)
  - Programs and data stored in memory
  - Instructions and data encoded as numbers (binary strings)

Components of a computer



CPU: control unit+data unit + registers

What's a computer? Summary

- Definition:

  "A computer is a digital device system that accepts digital information (binary data) and manipulates it according to a sequence of instructions known as a program"

- Classes of manipulation:
  - Communication
  - Processing
  - Storing

Programming a processor

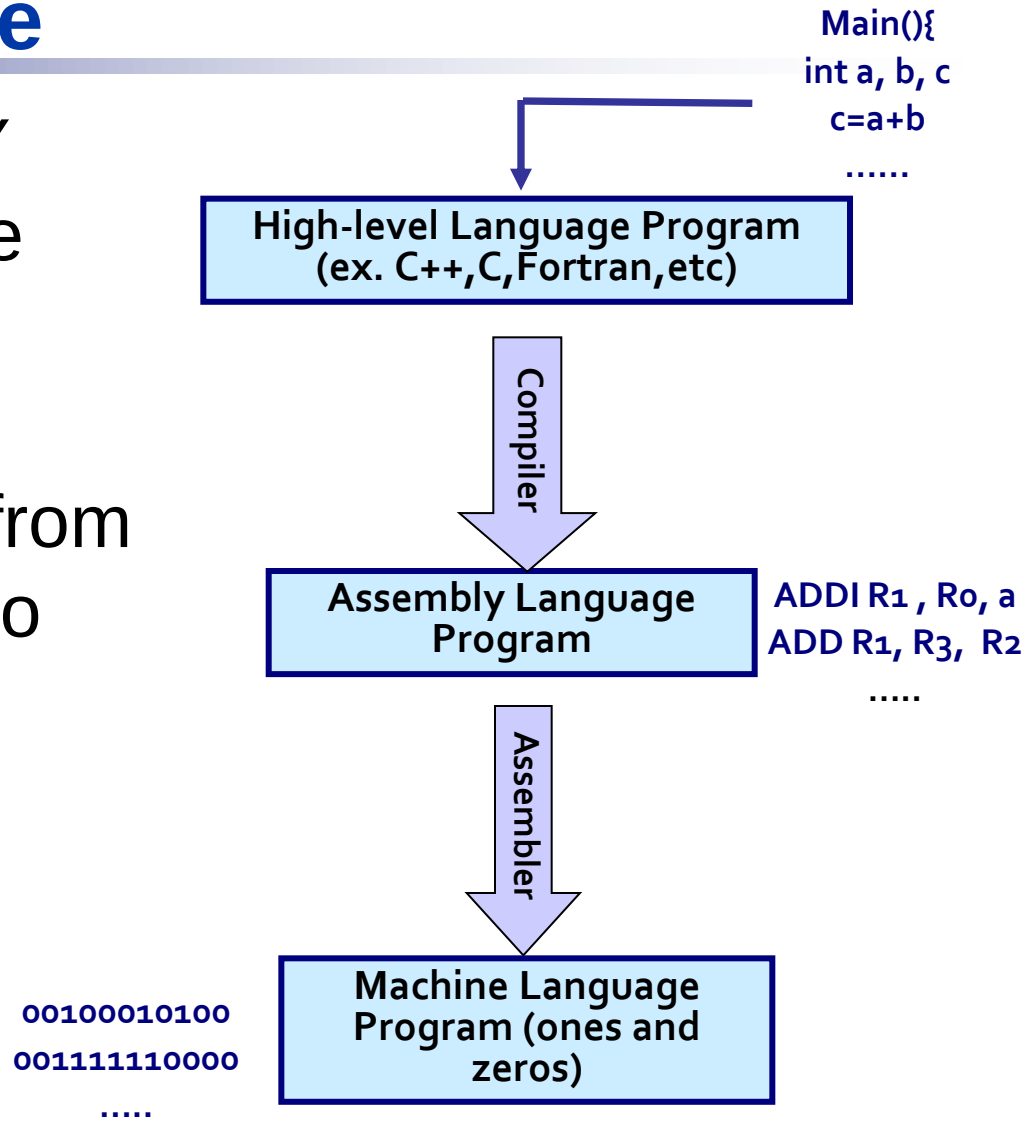
INTRODUCTION

Computer programming

- How to specify a task to a computer? ⟶ **PROGRAM**
- Program: algorithm written with a programming language
- Algorithm: procedure of well-defined steps to follow in order to solve a problem
- Programming language: syntax, rules and mnemonics designed to write algorithms that can be understood by a computer

From High-level language to hardware language

- The processor ONLY understands machine language (machine instructions)
- Translation process from high-level language to machine language



Classes of languages

- High-level programming language
  - Programming language with high semantic content instructions understandable by humans (C, C++, Java, Pascal,..)

```
#define loops 4
int sum = 0;
int V[4] = {0,1,2,3};
main () {
  register int partial_sum = 0;
  register int i = 0;
  while (i < loops) {
    partial_sum += V[i];
    i++;
  }
  sum = partial_sum;
}
```

Classes of languages

- Assembly language
  - Programming language with low semantic content understandable by humans (Intel 386, Alpha, MIPS,..)

```
.text
  main:
    MOVI R3,0            ; R3 is partial_sum
    MOVI R1,0            ; R1 is i
    MOVI R4, LO(V)
    MOVHI R4, HI(V)      ; R4 loads 1st elem. vector V
    MOVI R0,0            ; for jump instruction
    MOVI R2,Loops
  while:
    CMPLT R5,R1,r2       ; ¿i < Loops?
    BZ R5, fiwhile       ; if (R5==0) loop end
    ….
    ADDI R1,R1,1         ; i++
    ….
    BZ R0, while         ; jump to loop
Fiwhile:
```

  - Note: while, loop, main are labels, not syntaxis

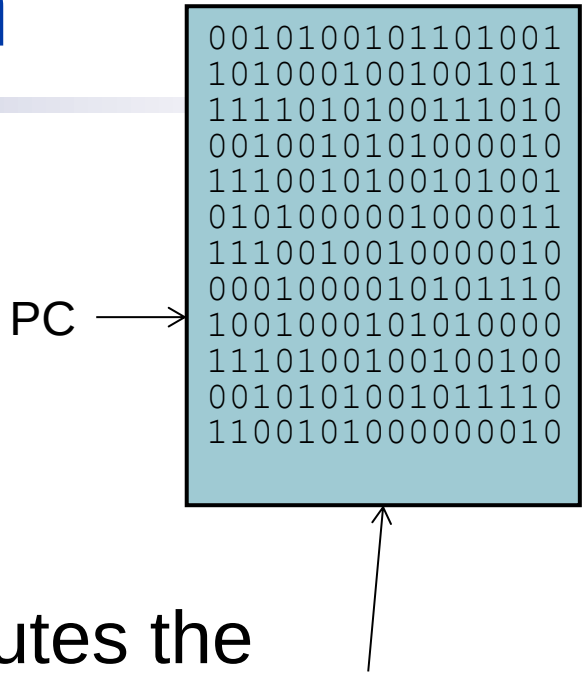
Classes of languages

- Machine language
  - Programming language with low semantic content understandable by the computer

Running a program

INTRODUCTION

Running a Program

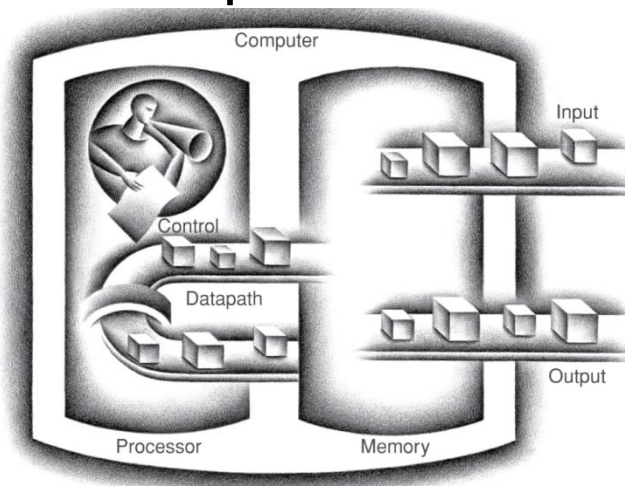


- The processor reads and executes the instructions of a program (stored in memory) sequentially
- Instruction execution steps (instruction cycle):
  - Fetching (from memory): the instruction is read from memory using the address in the PC (Program Counter)

Running a Program

- The processor reads and executes the instructions of a program (stored in memory) sequentially
- Instruction execution steps (instruction cycle):
  - Fetching (from memory): the instruction is read from memory using the address in the PC (Program Counter)
  - Decoding: the type of instruction and the operands are determined. What is an operand?

What to do on what



Running a Program

- The processor reads and executes the instructions of a program (stored in memory) sequentially
- Instruction execution steps (instruction cycle):
  - Fetching (from memory): instruction is read from memory using the address in the PC (Program Counter)
  - Decoding: type of instruction and the operands are determined
  - Execution: operation specified by the instruction is executed

Overview

- Introduction
  - What is a computer?
  - Programming a processor
  - Running a program
- Instruction set architecture
  - Instruction set, Pat-Hen 2.1-2.3
  - Instruction formats Pat-Hen 2.5
  - Addressing modes
- Assembly language programming 2.5-2.8, 2.10
  - Introduction to MIPS processor
  - MIPS assembly language
  - Translation of high-level structures to MIPS assembly language

Instruction set architecture

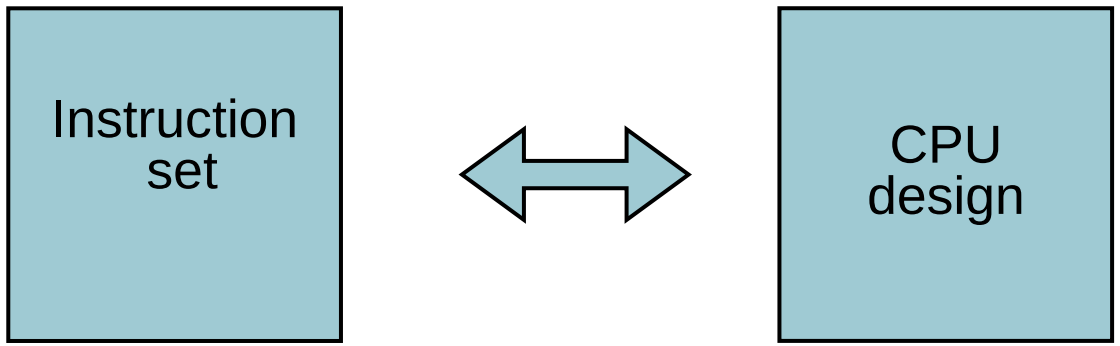
INSTRUCTION SET ARCHITECTURE

Instruction Set Architecture

- ISA (or architecture): abstract interface between hardware and lowest level software
- format of communication: what does an instruction look like
  - Part of the processor visible to programmer or compiler
- ISA can be described as the collection of:
  - Instruction operands: implicit, explicit, in memory
    - Number of operands
  - Type of data
  - Number and size of registers
  - Type and size of instructions
  - …

Instruction set

- A processor is designed to “understand” an instruction set
  - The repertoire of instructions of a computer
  - Different computers have different instruction sets



Topic 1

Topic 2

Instruction set

- Instruction categories:
  - Data Transfer: transfer data between memory and registers or memory and I/O ports

    to get data, to store results (hidden on higher level)
  - Computation: computational operations on data
    - Arithmetical, logical shifts, comparison,…

    The real work
  - Control: control the flow of the program

    influence the program counter (PC) to get IF, do-loops, goto working

Instruction set

CPU design

- Example of processor: r-MIPS
  - Data Transfer:
    - LW (load word)
    - SW (store word)
    - LUI (load upper immediate)
  - Computation:
    - ADD (addition), SUB (subtraction), ADDI (add immediate)
    - OR (logical or), AND (logical and), ORI (or immediate)
    - SRL (shift right logical), SLL (shift left logical) SRA (shift right arithmetical)
    - SLT (comparison, set less than)
  - Control:
    - BEQ (branch on equal)
    - J (unconditional jump)

Instruction set

Instruction formats

INSTRUCTION SET ARCHITECTURE (2.5)

Instruction format

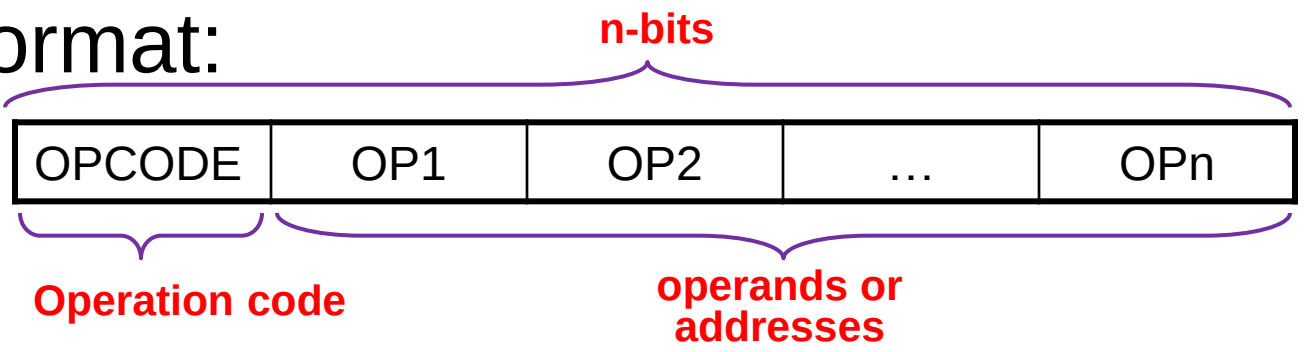
- The instructions are stored in memory → Binary encoding (n-bits)
  - ADD A, B, C → 0100100100100101

Instruction format

- The instructions are stored in memory → Binary encoding (n-bits)
  - ADD A, B, C → 0100100100100101
- The bit-string is divided into fields. What is a field?
- Instruction fields
  - Operation code (OPCODE): specifies the operation (ADD)
  - Operand (addresses): gives operands needed to execute the instruction (A, B, C)

Instruction format

- The instructions are stored in memory → Binary encoding (n-bits)
  - ADD A, B, C → 0100100100100101
- Instruction fields:
  - Operation code (OPCODE): specifies the operation (ADD)
  - Operand (addresses): operands needed to execute the instruction (A, B, C)
- Format:



Instruction format

- OPCODE
  - Type of operation encoded in binary
  - Variable or fixed format (?)

Instruction format

- OPCODE
  - Type of operation encoded in binary
  - Variable or fixed format
- Operands or addresses
  - Specify the source and destination operands of the instruction (binary encoded)
    - Constant, register, memory address

Instruction format

- OPCODE
  - Type of operation encoded in binary
  - Variable or fixed format
- Operands or addresses
  - Specify the source and destination operands of the instruction (binary encoded)
    - Constant, register, memory address
  - There are explicit and implicit operands (?)
    - M-address machine: the instruction format allows at most M explicit operands

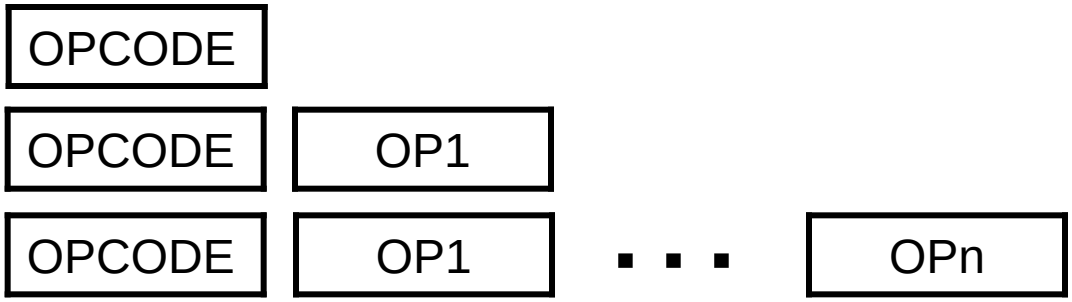
Instruction format

- OPCODE
  - Type of operation encoded in binary
  - Variable or fixed format
- Operands or addresses
  - Specify the source and destination operands of the instruction (binary encoded)
    - Constant, register, memory address
  - There are explicit and implicit operands (?)
    - M-address machine: the instruction format allows at most M explicit operands
  - Addressing mode: indicates how to interpret the operand field
    - Each operand has its own addressing mode

Instruction format

- Instruction format of fixed length
  - All instructions of the same length (number of bits) Why would you do so?

- Instruction format of variable length
  - Instruction length varies
  - Several memory words



Addressing modes

INSTRUCTION SET ARCHITECTURE

Addressing modes

- How are the m-bits operand field interpreted to obtain the operand?
- Instruction looks like

ADD A, B, C → **0100100100100101** **(A ← B + C)**

  - What is the interpretation, what does it do?
  - Where goes the value of **A**?
  - What specific data must be added to **C** to calculate the value to be stored in **A**?
  - The addressing mode of operand **B** gives the meaning of **0010**

The addressing mode is included in the OpCode

Addressing modes (I)

Opcode specifies how to interpret the operand

- Immediate addressing
- Direct addressing
  - Absolute
    - Register mode
    - Memory mode
  - Base (Relative or Displacement) addressing
    - Implicit register (Base, PC, index, SP,…)
    - Explicit register
- Indirect addressing
  - Register mode
  - Memory mode

Addressing modes (II)

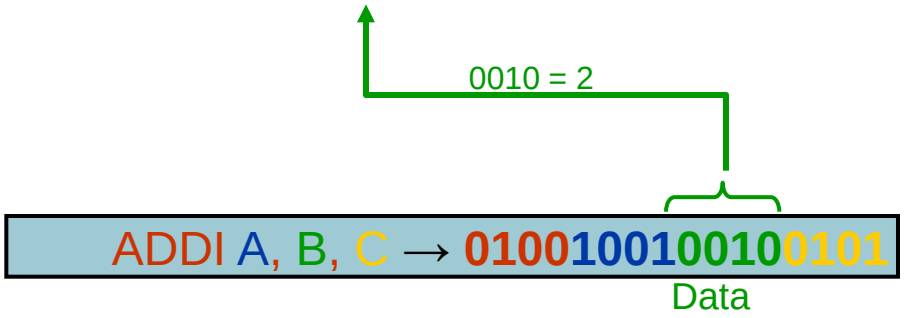
- Immediate addressing, let us call ADDI
  - Operand field contains value of the operand
  - So, operand value is inside the instruction
  - Type of data (natural, integer, real,…) binary encoded
  - Example: B is a natural integer operand
    - **A = ? + C**

ADDI A, B, C → **0100100100100101**

How many integer values can be represented by the B field ?

Addressing modes (II)

- Immediate addressing
  - Type of data (natural, integer, real,…) binary encoded
  - Example: B is an integer operand
    - **A = 2 + C**



So data hidden in instruction memory, not in data memory

Addressing modes (III)

- Direct addressing
  - Operand field indicates WHERE operand value can be found
  - Operand value can be in a register or in a memory location
  - Two different ways to indicate where the operand is:
    - ABSOLUTE: complete address of the operand is
      BASE (relative): partial information on the operand address $\Rightarrow$ the operand address needs to be calculated

How many addresses can be reached?

Addressing modes (IV)

- Direct Absolute addressing example opcodes
  - Register mode ADD2: the operand value in a register (binary code of the register address)
  - Memory mode ADD3: the operand is in a memory location (memory address)
- Ex. Register mode
  - A = ? + C

Memory

0000	0010
0001	0100
0010	0000
…	…
1111	0101

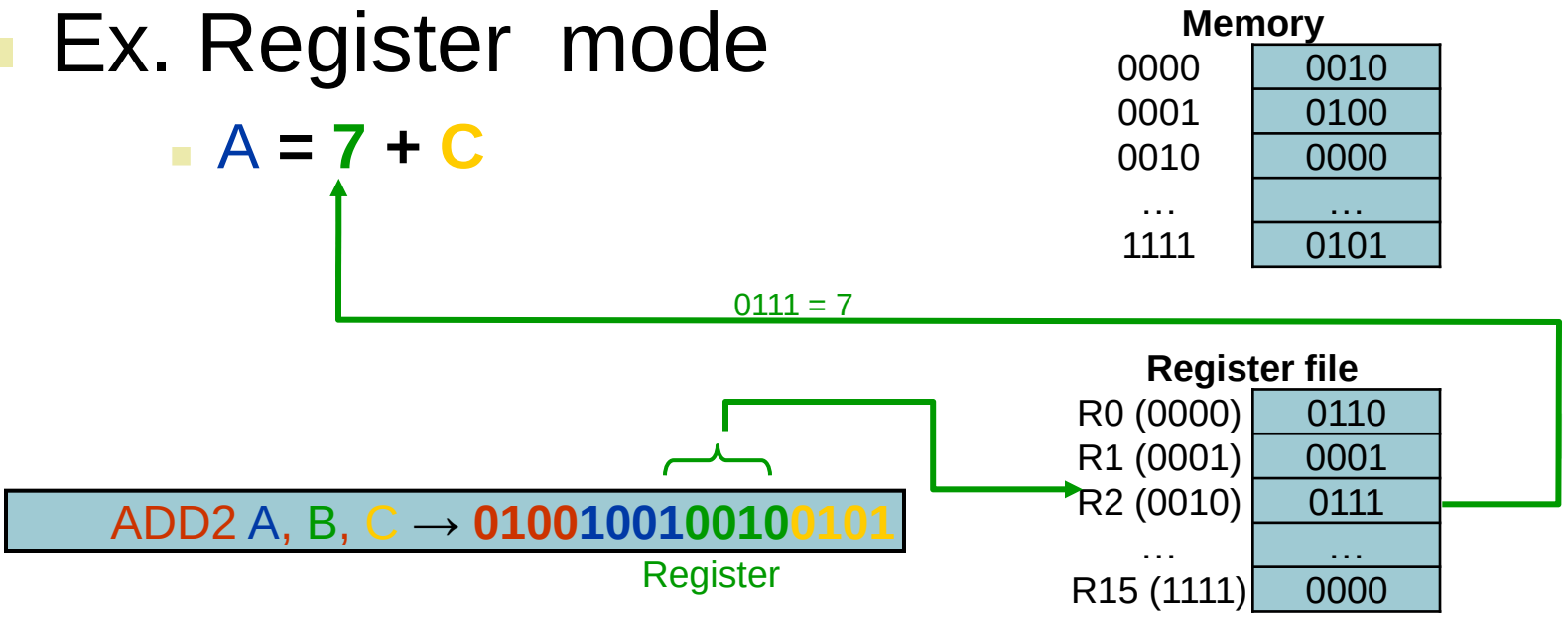
Register file

R0 (0000)	0110
R1 (0001)	0001
R2 (0010)	0111
…	…
R15 (1111)	0000

ADD2 A, B, C → **0100100100100101**

Addressing modes (IV)

- Direct Absolute addressing
  - Register mode ADD2: the operand is in a register (binary code of the register address)
  - Memory mode: the operand is in a memory location (memory address)
- Ex. Register mode
  - A = 7 + C



Addressing modes (IV)

- Direct Absolute addressing
  - Register mode: operand value in register (binary code of the register address)
  - Memory mode ADD3: operand value in memory location (memory address)
- Ex. Memory mode
  - A = ? + C

Memory

0000	0010
0001	0100
0010	0000
…	…
1111	0101

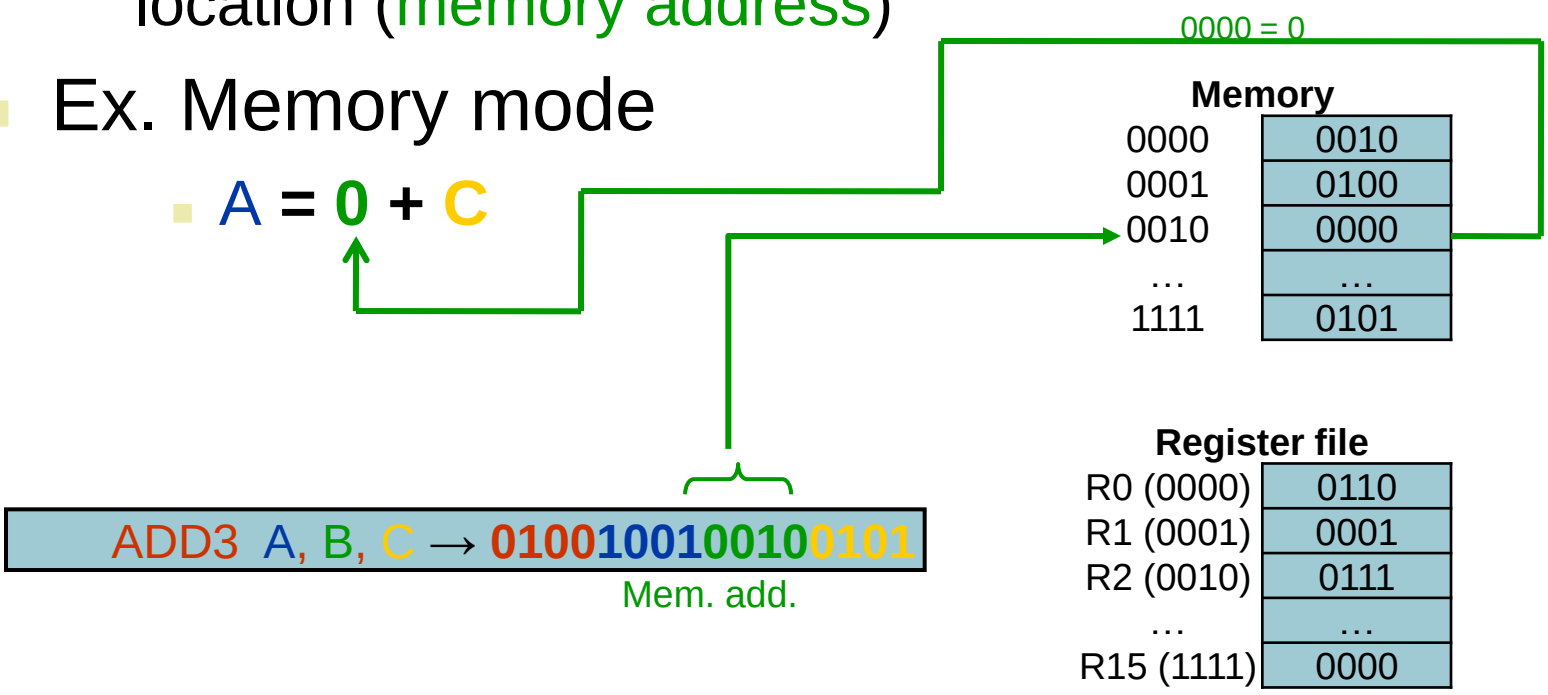
Register file

R0 (0000)	0110
R1 (0001)	0001
R2 (0010)	0111
…	…
R15 (1111)	0000

ADD3 A, B, C → 0100100100100101

Addressing modes (IV)

- Direct Absolute addressing
  - Register mode: operand value in register (binary code of the register address)
  - Memory mode ADD3: operand value in memory location (memory address)
- Ex. Memory mode
  - A = 0 + C



How many addresses can be reached?

Addressing modes (V)

Increase the address range by

Base addressing

- Operand value in a memory location
- Effective address (EA): operand memory address
- EA is the sum of offset (displacement) and base which can be found in a register
  - Explicit register: register and offset are specified in the instruction, each in a field
  - Implicit register: the register is not specified in the instruction, for instance Program Counter (PC)

Addressing modes (V)

- Base addressing with implicit register:
  - The register is not specified in the instruction format
  - Only the offset is specified
    - Relative to PC (Program Counter), Relative to a Base, Relative to an Index, Relative to SP (Stack Pointer) …
- Ex. Relative to a Base
  - A = ? + C

Memory

0000	0010
0001	0100
0010	0000
…	…
1111	0101

Base

1101

Thinking in unsigned terms; addresses are positive

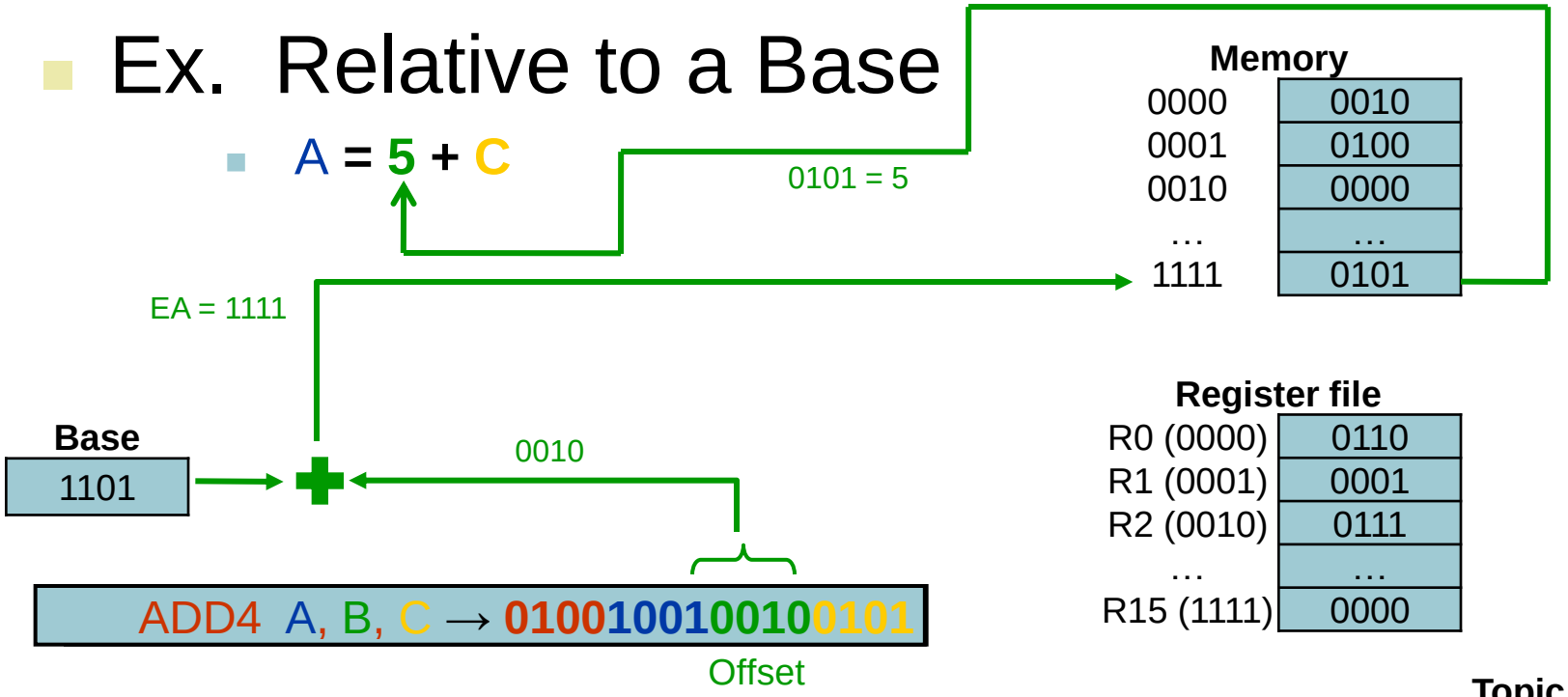
Register file

R0 (0000)	0110
R1 (0001)	0001
R2 (0010)	0111
…	…
R15 (1111)	0000

ADD4 A, B, C → 0100100100100101

Addressing modes (V)

- Base addressing with implicit register:
  - The register is not specified in the instruction format
  - Only the offset is specified
    - Relative to PC (Program Counter), Relative to a Base, Relative to an Index, Relative to SP (Stack Pointer) …
- Ex. Relative to a Base
  - A = 5 + C



Addressing modes (V)

- Base addressing with explicit register:
  - register with base value explicitly specified in instruction format
  - Two fields in the instruction format associated with operand: register with base value and offset
- Ex.
  - A = ? + C

Memory

0000	0010
0001	0100
0010	0000
…	…
1111	0101

Register file

R0 (0000)	0110
R1 (0001)	0001
R2 (0010)	0111
…	…
R15 (1111)	0000

ADD5 A, B(D), C → **0100100111110001**0101

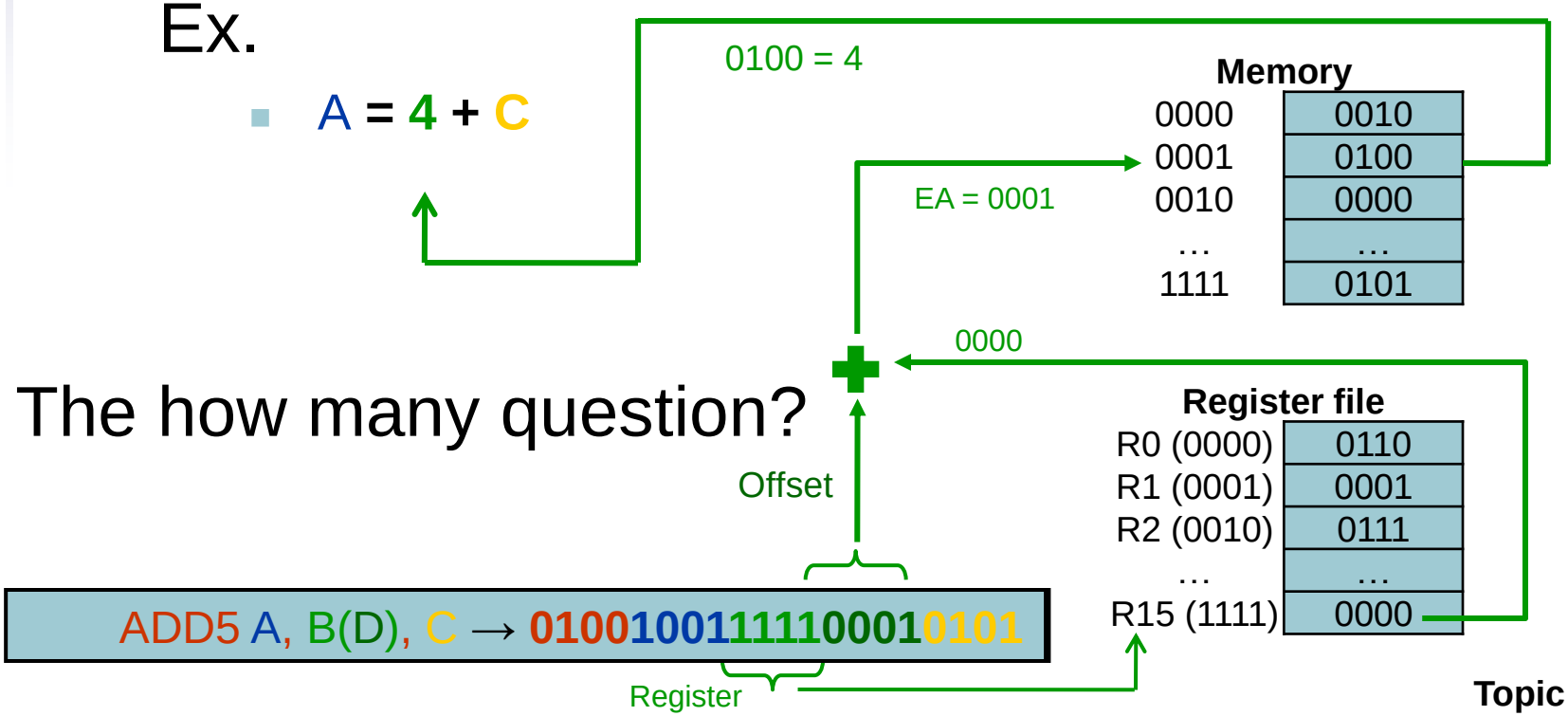
Register Offset

Addressing modes (V)

- Base addressing with explicit register:
  - register with base value explicitly specified in the instruction format
  - Two fields in the instruction format associated with the operand: register with base value and offset

Ex.

- A = 4 + C



The how many question?

Addressing modes (V)

- Base addressing why is this useful?
  - Possibility to walk through a set of memory addresses with one instruction

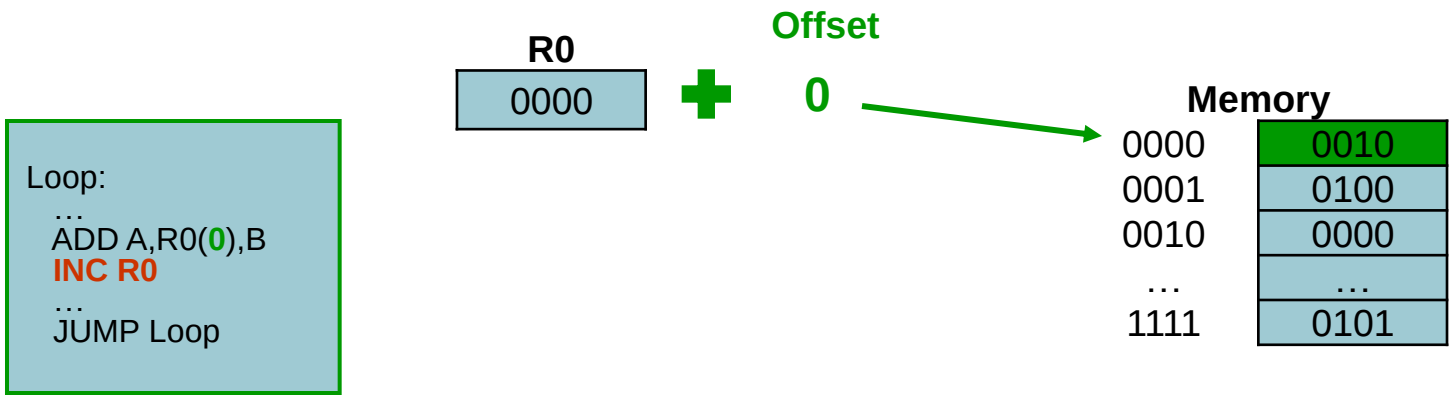
```
Loop:
   …
   ADD A,R0(0),B
   INC R0
   …
   JUMP Loop
```

Memory

0000	0010
0001	0100
0010	0000
…	…
1111	0101

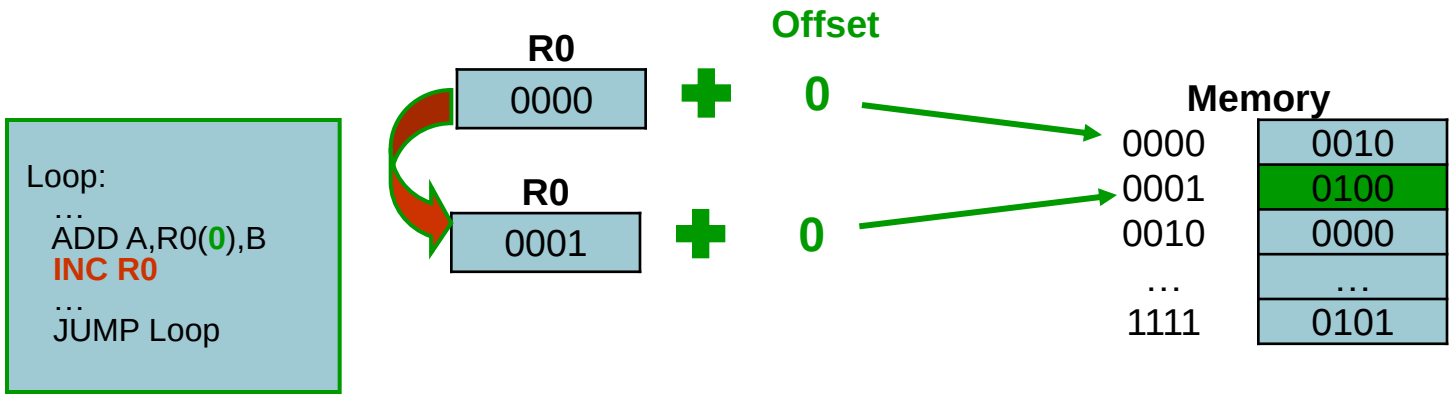
Addressing modes (V)

- Base addressing why is this useful?
  - Possibility to walk through a set of memory addresses with one instruction



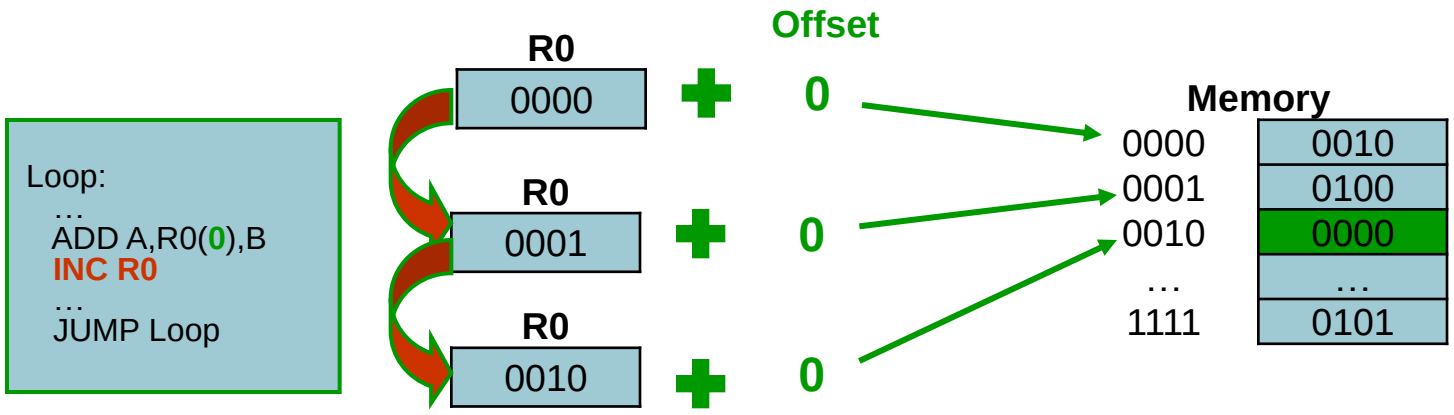
Addressing modes (V)

- Base addressing why is this useful?
  - Possibility to walk through a set of memory addresses with one instruction



Addressing modes (V)

- Base addressing why is this useful?
  - Possibility to walk through a set of memory addresses with one instruction



What does this example do?

Addressing modes (V)

- Auto-indexed addressing
  - Automatic increase/decrease of register content
  - Useful in loops like the last one (instruction INC is not needed anymore)
  - ADD A, R0(0),B
    - Pre-autoincrease: R0=R0+1; EA=R0+0;
    - Pre-autodecrease: R0=R0-1; EA=R0+0;
    - Post-autoincrease: EA=R0+0; R0=R0+1;
    - Post-autodecrease: EA=R0+0; R0=R0-1;

Addressing modes (VI)

- Indirect addressing
  - Operand field specifies WHERE is the operand ADDRESS
  - Operand value in memory location
  - Location address can be specified by:
    - Register: operand field specifies register address
    - Memory: operand field specifies memory address

So we reach Da Vinci code level. Given address in field we reach a box that contains the address of where the value may be hidden.

Addressing modes (VII)

- Register indirect addressing:
  - The instruction format specifies the register file address where the memory address of the operand is stored
- Example:
  - A = ? + C

Memory

0000	0010
0001	0100
0010	0000
…	…
1111	0101

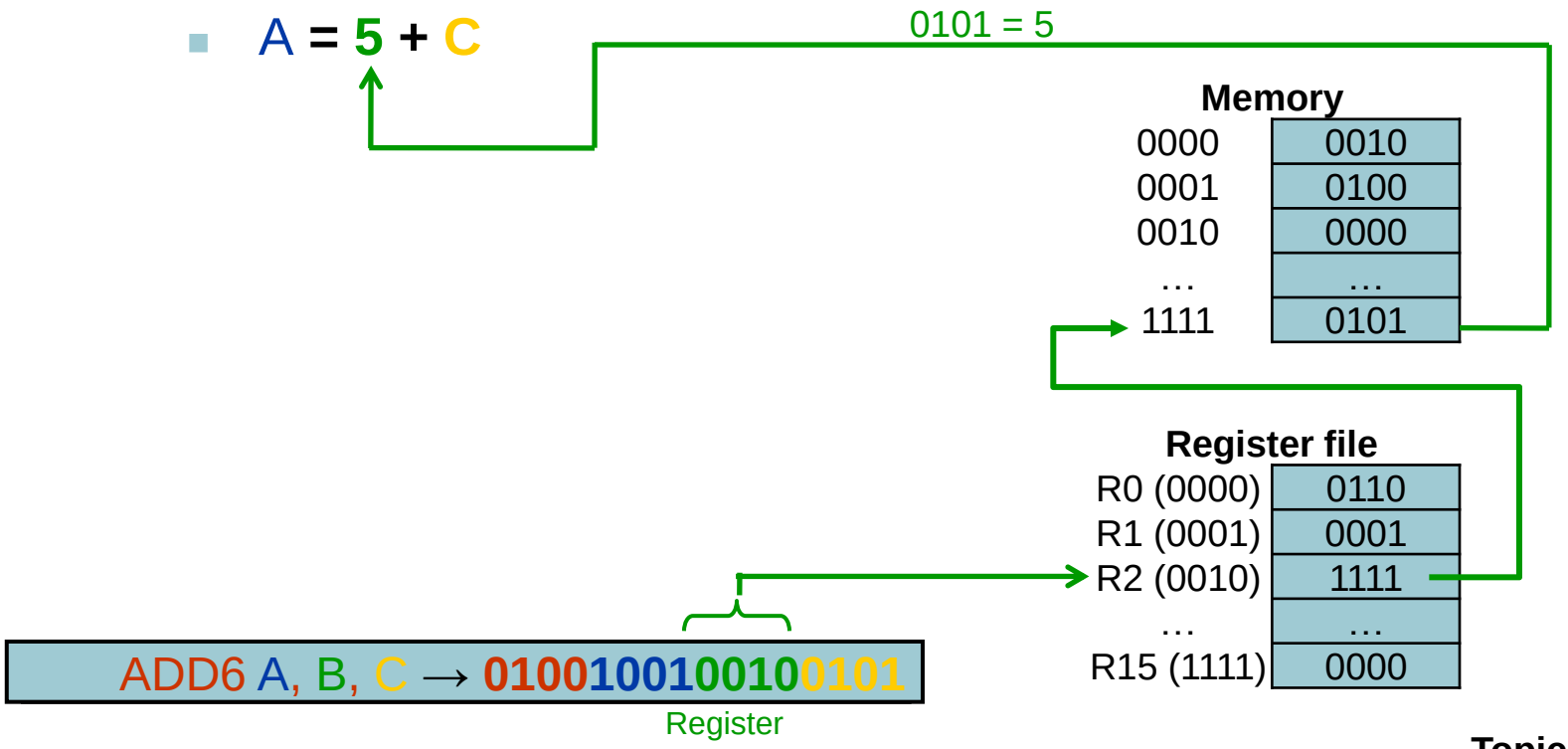
Register file

R0 (0000)	0110
R1 (0001)	0001
R2 (0010)	1111
…	…
R15 (1111)	0000

ADD6 A, B, C → **0100100100100101**

Addressing modes (VII)

- Register indirect addressing:
  - The instruction format specifies the register file address where the memory address of the operand is stored
- Example:
  - A = 5 + C



Addressing modes (VII)

- Memory indirect addressing:
  - The instruction format specifies the memory address where the memory address of the operand is stored
- Example:
  - A = ? + C

Memory

0000	0010
0001	0100
0010	0000
…	…
1111	0101

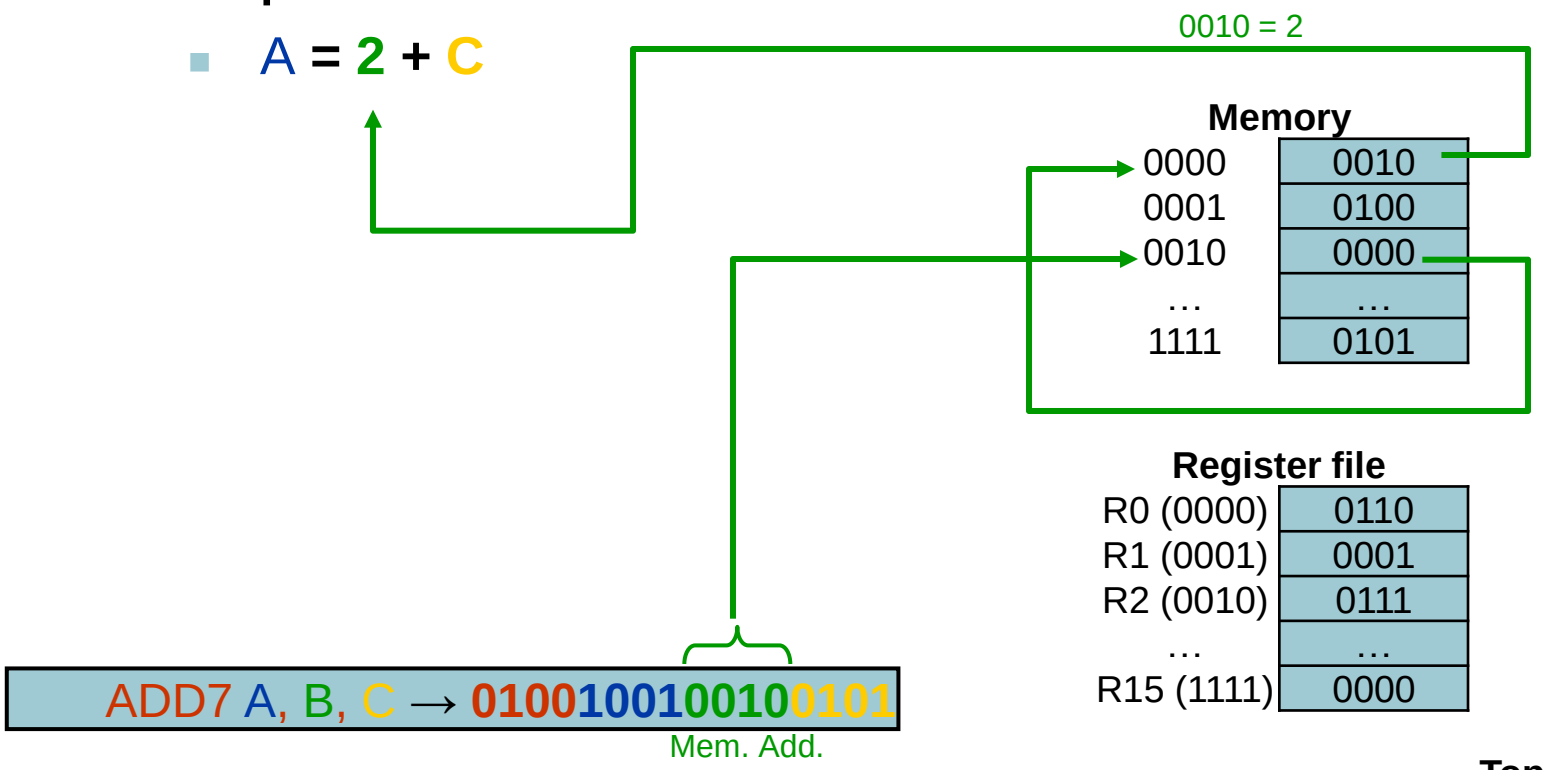
Register file

R0 (0000)	0110
R1 (0001)	0001
R2 (0010)	0111
…	…
R15 (1111)	0000

ADD7 A, B, C → **0100100100100101**

Addressing modes (VII)

- Memory indirect addressing:
  - The instruction format specifies the memory address where the memory address of the operand is stored
- Example:
  - A = 2 + C



Instruction format example

- Design the instruction format for a processor with the following features:
  - 64 instructions of 32 bits.. Name size in bits?
  - 32 registers of 32 bits.. Address size?
  - 3 addressing modes: Register, direct memory and PC-relative addressing.. Mode indicator?
  - Instructions with 2 operands:
    - 1st operand field specifies a register Field sizes?
    - 2nd operand field specifies a memory location

Instruction format example

- Design the instruction format for a processor with the following features:
  - 64 instructions of 32 bits
  - 32 registers of 32 bits
  - 3 addressing modes: Register, direct memory and PC-relative addressing
  - Instructions with 2 operands:
    - 1st operand specifies a register
    - 2nd operand specifies a memory location

With name of the instruction

Instruction format example

- Design the instruction format for a processor with the following features:
  - 64 instructions of 32 bits **=> 64 = 2^6 => 6 bits (OPCODE)**
  - 32 registers of 32 bits
  - 3 addressing modes: Register, direct memory and PC-relative addressing
  - Instructions with 2 operands:
    - 1st operand specifies a register
    - 2nd operand specifies a memory location

Instruction format example

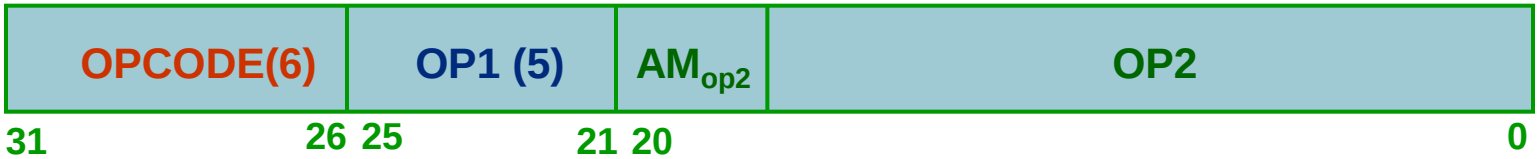
- Design the instruction format for a processor with the following features:
  - 64 instructions of 32 bits **=> 64 = 2^6 => 6 bits (OPCODE)**
  - 32 registers of 32 bits
  - 3 addressing modes: Register, direct memory and PC-relative addressing
  - Instructions with 2 operands:
    - 1st operand specifies a register
    - 2nd operand specifies a memory location

Instruction format example

- Design the instruction format for a processor with the following features:
  - 64 instructions of 32 bits **=> 64 = 2^6 => 6 bits (OPCODE)**
  - 32 registers of 32 bits
  - 3 addressing modes: Register, direct memory and PC-relative addressing
  - Instructions with 2 operands:
    - 1st operand specifies a register => 32 = 2^5 => 5 bits (OP1)
    - 2nd operand specifies a memory location

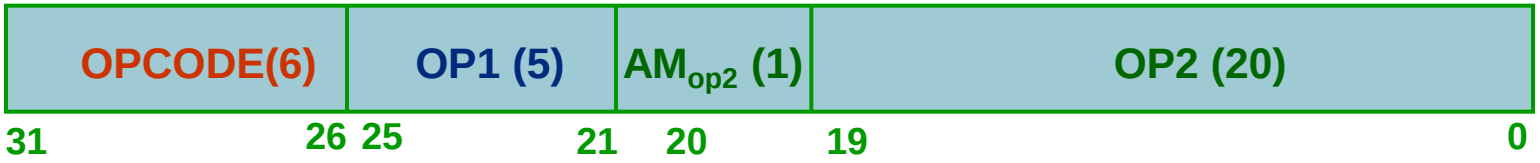
Instruction format example

- Design the instruction format for a processor with the following features:
  - 64 instructions of 32 bits **=> 64 = 2^6 => 6 bits (OPCODE)**
  - 32 registers of 32 bits
  - 3 addressing modes: Register, direct memory and PC-relative addressing
  - Instructions with 2 operands:
    - 1st operand specifies a register => 32 = 2^5 => 5 bits (OP1)
    - 2nd operand specifies a memory location



Instruction format example

- Design the instruction format for a processor with the following features:
  - 64 instructions of 32 bits **=> 64 = 2^6 => 6 bits (OPCODE)**
  - 32 registers of 32 bits
  - 3 addressing modes: Register addressing, direct memory addressing and PC-relative
  - Instructions with 2 operands: $2 = 2^1 \Rightarrow 1$ bit (AM_{op2})
    - 1st operand specifies a register => $32 = 2^5$ => 5 bits (OP1)
    - 2nd operand specifies a memory location



You speak hex?
Me, not entender.

I say, 1100 0111
What do you have to say to that?
¿Que?

If you say *a*, you also should say *b*

Oh, you mean from 1010 follows 1011?

Do you speak hexadecimal?
Yes, I do, piece of cake.

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

Example: eca8 6420

- 1110 1100 1010 1000 0110 0100 0010 0000

Overview

- Introduction
  - What is a computer?
  - Programming a processor
  - Running a program
- Instruction set architecture
  - Instruction set
  - Instruction set, Pat-Hen 2.1-2.3
  - Instruction formats Pat-Hen 2.5

Assembly language programming 2.5-2.8, 2.10

- Introduction to MIPS processor
- MIPS assembly language
- Translation of high-level structures to MIPS assembly language

Introduction to MIPS processor

ASSEMBLY LANGUAGE PROGRAMMING

Introduction to MIPS processor (I)

- MIPS (*Microprocessor without Interlocked Pipeline Stages*): family of RISC microprocessors developed by MIPS Technologies
- 32-bits processor:
  - 32-bits registers, 32-bits ALU operands, 32-bits bus width *What is a bus width?
- 32 general purpose registers *Address size?
  - Available to the programmer
  - $0 value is always 0
- Program Counter (PC) is 32 bits wide
- All MIPS instructions are 32 bits long (1 word)

*How many instructions can a program have?

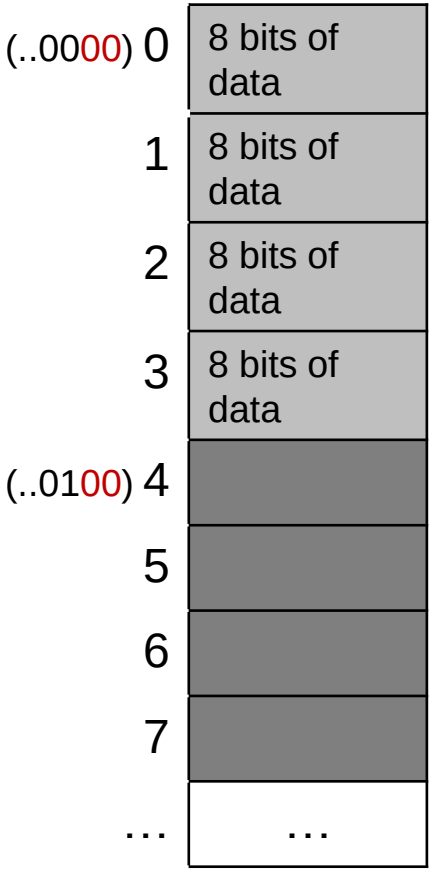
MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	no
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

Introduction to MIPS processor (II): Memory organization

- 4 byte words
- Byte addressing:
  - Two consecutive words are 4 positions apart
  - All instructions are in memory addresses multiple of 4 (two least significant bits 00)
  - Instructions are in aligned locations: PC (32-bits) is increased by 4
- Number of bytes in memory: 2^{32}
- Number of words in memory: $2^{32}/4=2^{30}$
- What about shift left logical, sll

  011 → 0110 → 011**00**



MIPS (RISC) Design Principles

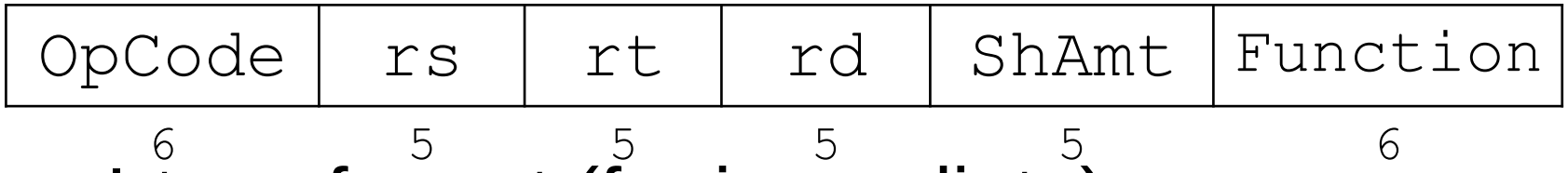
- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes

How many you said?

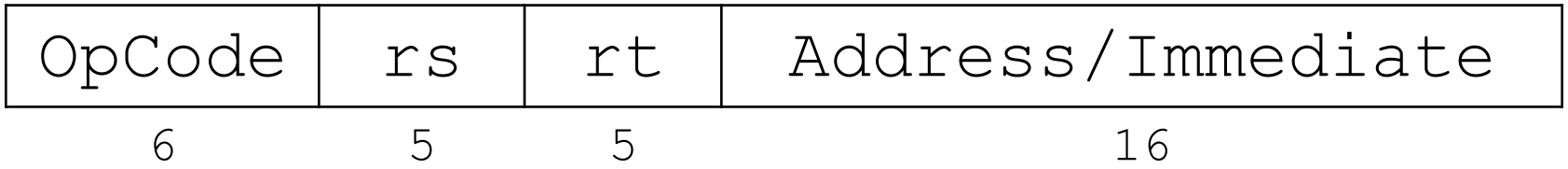
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands
- Good design demands good compromises
  - three instruction formats

MIPS Instruction formats

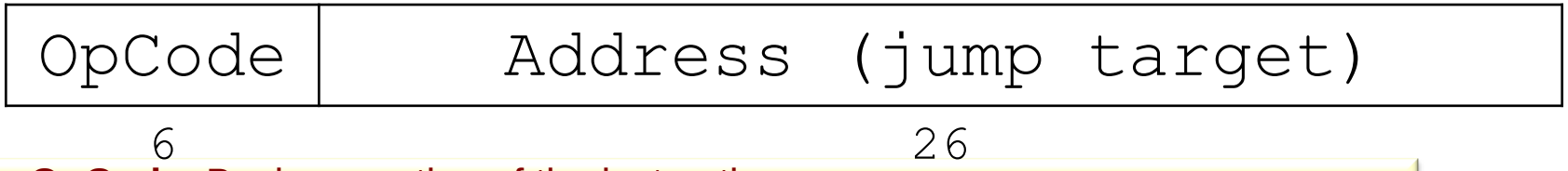
- R-type format (for register)



- I-type format (for immediate)



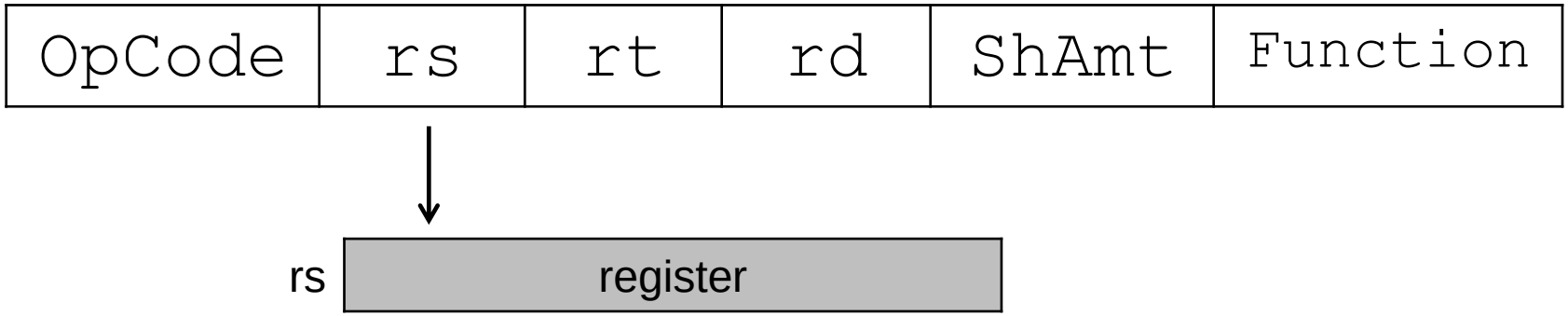
- J-type format (for jump)



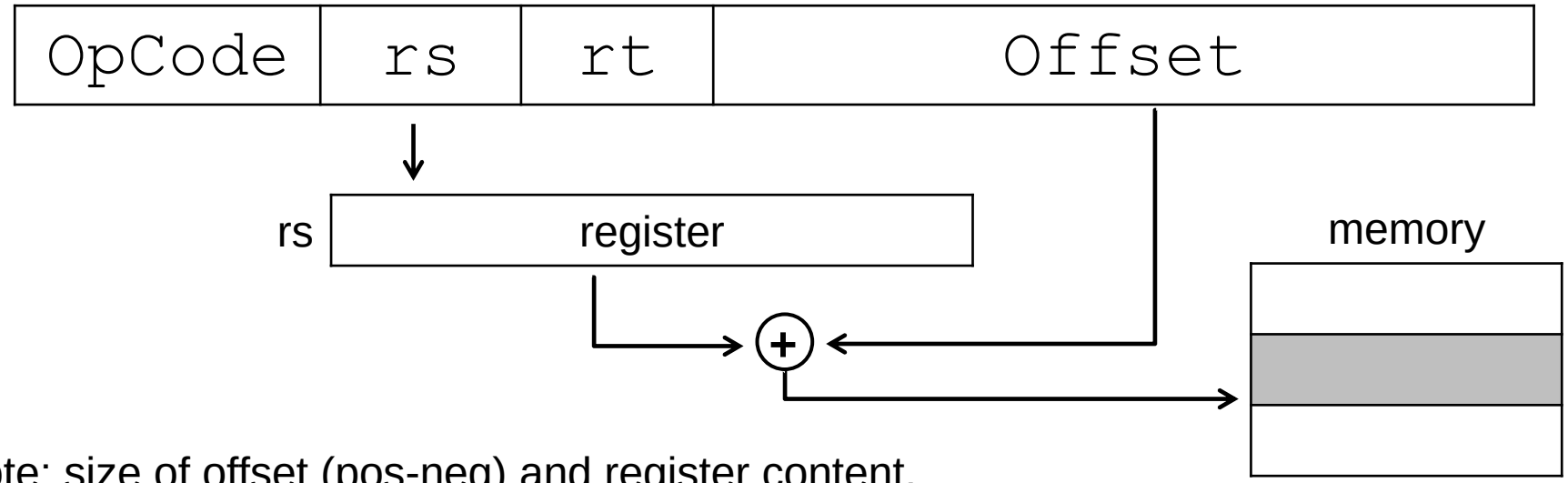
- **OpCode**: Basic operation of the instruction
- **rs**: register file address of first source operand, 1st operand
- **rt**: register file address of second source operand, 2nd operand
- **rd**: register file address of result destination, operand result
- **ShAmt**: Shift Amount (for shift instructions)
- **Function**: selects the specific variant of the operation in the OpCode field

MIPS Addressing modes

- Register addressing



- Base relative addressing



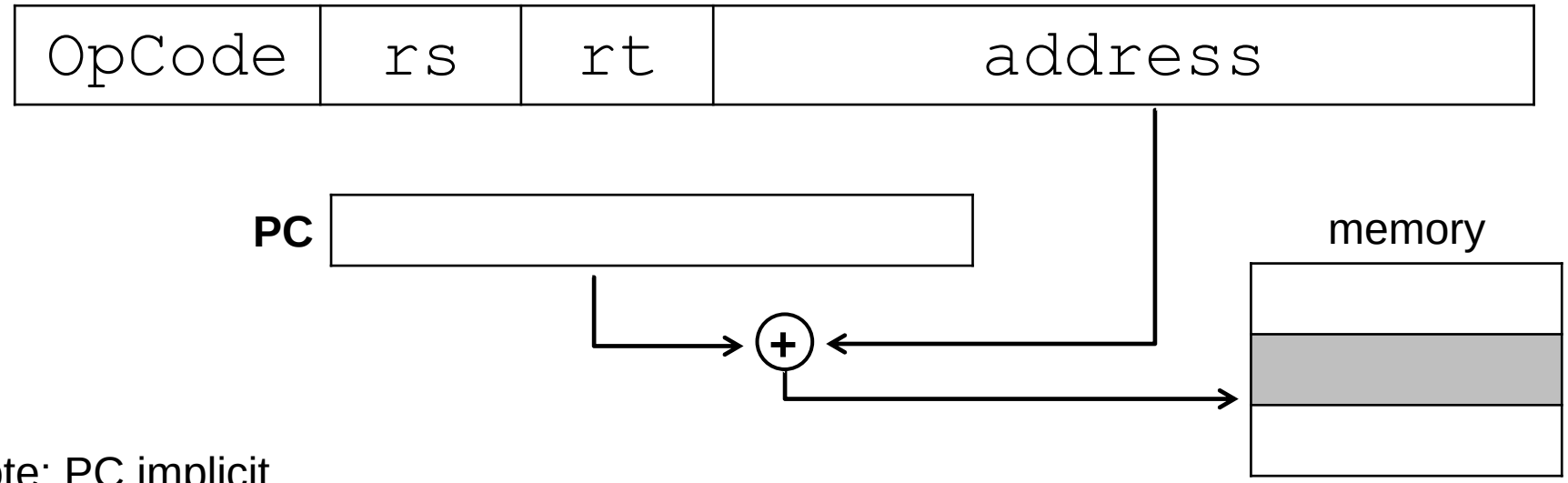
Note: size of offset (pos-neg) and register content.

MIPS Addressing modes

- Immediate addressing

Note: extend the immediate

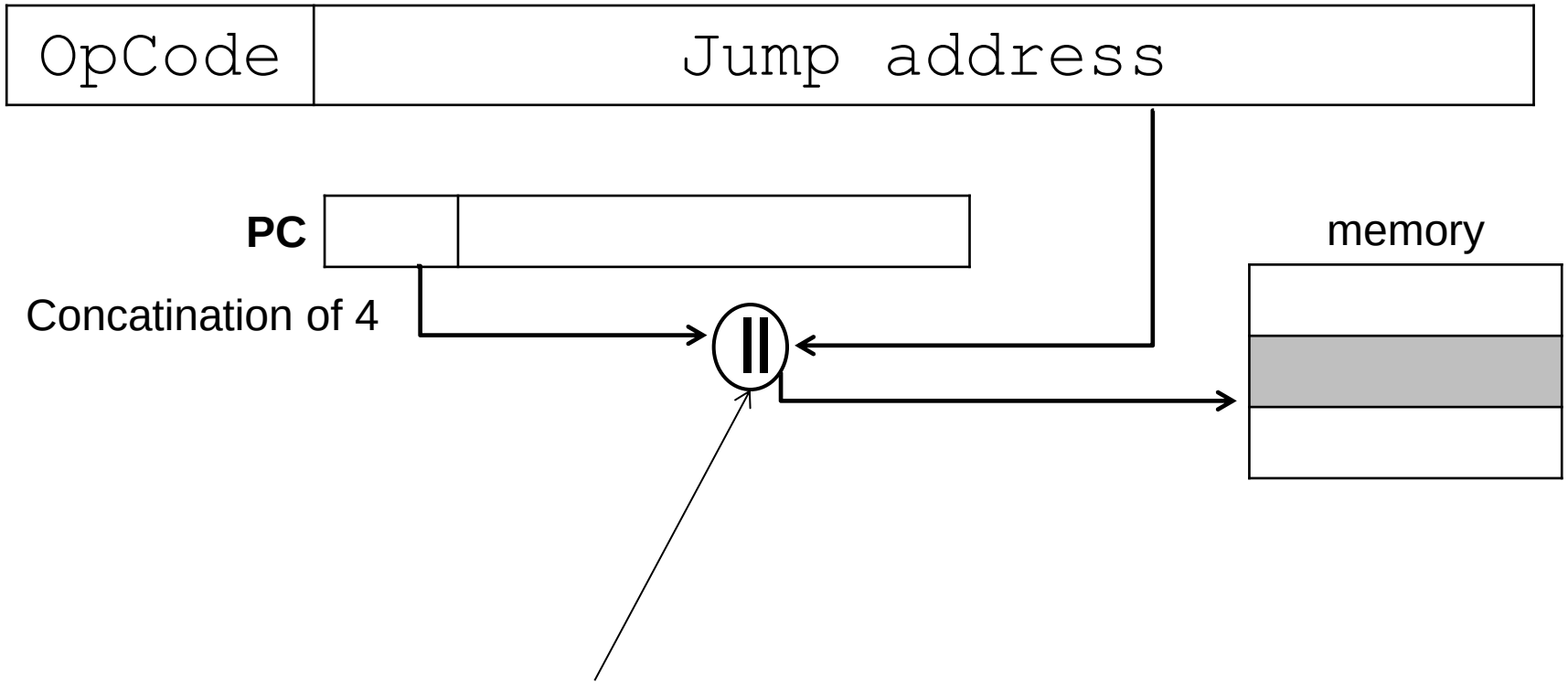
- PC-relative addressing



Note: PC implicit

MIPS Addressing modes

- Pseudo-direct addressing



Hm, what will this strange symbol mean?
How does it work with the bits here?

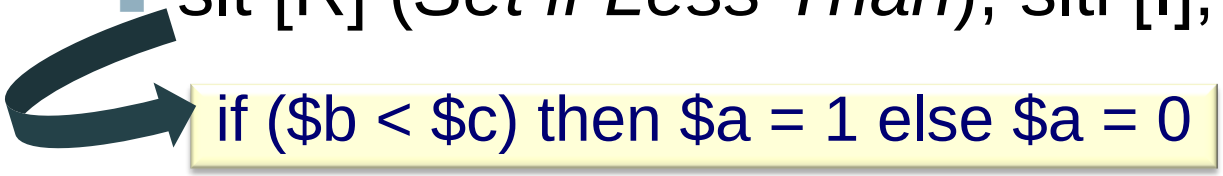
Instruction set classification

- Arithmetical-logical instructions (50% of instructions): add [R], sub [R], addi [I], ori [I], ...
- Transfer data instructions (30-40% of instructions)
  - Load in register: lw [I] (*Load Word*), lb [I] (*Load Byte*), lui [I] (*Load Upper Immediate*)
  - Store in memory: sw [I] (*Store Word*), sb [I] (*Store Byte*)

lw – the destination is first operand.
sw – the destination is second op

Instruction set classification

- Conditional branch instructions (10-20%)
  - beq [I] (*Branch on Equal*) , bne [I] (*Branch on Not-Equal*)
  - slt [R] (*Set if Less Than*), slti [I], …



- Unconditional branch instructions (instruction frequency: 1-2%)
  - j [J] (*Jump*), jr [R] (*Jump to Register*), jal [J] (*Jump and link*)

MIPS Arithmetic instructions

- Usually, R-type format instructions:
  - The 6-bits `OpCode` field for R instructions is `000000`
  - Specific operation indicated by the 6-bits `function` field
  - Two source registers (`rs` and `rt`), result stored in destination register `rd`
  - 5-bits `shamt` field to specify the amount to be shifted in arithmetical or logical shift

  Why is the latter 5 bits?

Arithmetic-logical instructions
Ex.: add

Opcode	rs	rt	rd	Shamt	Funct
000000	00111	01001	00011	00000	100000
OpALU	Reg 7	Reg 9	Reg 3		sum

Format: R-type

Meaning: Depends on the Funct field

Example: add content register 7 ($7) to content 9 ($9), store result into register 3 ($3): $3←$7+$9

Assembly code: `add $3, $7, $9`

Others: `sub, and, or, slt ...`

Arithmetic-logical instructions
Ex.: addi

Opcode	rs	rt	immediate
001000	00111	00011	0000000000001111
Sum	Reg 7	Reg 3	Constant Oper.=15

Format: I-type

Meaning: rt←rs + immediate

Example: add constant value 15 to content of register 7 ($7) and store the result into register 3 ($3): $3←$7+15

Assembly code: `addi $3, $7, 15`

Use arithmetic operations to calculate expressions

- Code C:
  - f = (g+h)-(i+j);
- Register use:
  - f →$16, g→$17, h→$18, i→$19, j→$20, tmp1→$8, tmp2→$9
- Assembly code:

Use arithmetic operations to calculate expressions

- C code:
  - f = (g+h)-(i+j);
- Register use:
  - f →$16, g→$17, h→$18, i→$19, j→$20, tmp1→$8, tmp2→$9
- Assembly code:

```
add $8, $17, $18   #tmp1←g+h
add $9, $19, $20   #tmp2←i+j
sub $16, $8, $9    #f←tmp1-tmp2
```

MIPS Data transfer instructions

- MIPS has only 32 registers, few data
- Data in memory → memory2register and register2memory transfers
- Usually, I-type format:
  - Load word (32 bits), half word or byte from memory to register (`lw`, `lh`, `lb`,...)

    store (`sw`, `sh`, `sb`,...) register to memory,
  - Register addressing: `rt`
  - Base addressing (relative): `Mem[rs+offset]`
  - Other: load constant into register (`lui`,...)

Data transfer instructions
Example sw

Opcode	rs	rt	immediate
101011	00111	00011	0000000000001100
Store W	Reg 7	Reg 3	offset=12

Format: I-type

Meaning: Mem[rs+offset]←rt

Example: stores content of register 3 ($3) in memory location addressed as sum of content register 7 ($7) and offset 12: Mem[$7+12]←$3

Assembly code: `sw $3, 12($7)`

sw- destination in first operand

How many memory addresses can be reached?

Data transfer instructions
Example lui

Opcode	rs	rt	immediate
001111	00000	00011	0000000000001111
Load I		Reg 3	constant=15

Format: I-type

Meaning: rt←constant*2^{16}

Example: stores the constant value 15 in the 16 most significant bits of register 3 ($3): $3 ← 15*$2^{16}$

Assembly code: `lui $3, 15`

destination in second operand

What is upper?

What is most significant bits?

Used to access vectors/arrays

- C code:
  - g=h+A[i]
- Registers and memory use:
  - g→$17, h→$18, i→$19, tmp1→$8, Astart→ starting memory address of A
- Assembly code:

Use: to access to vectors/arrays

- C code:
  - g=h+A[i]
- Registers and memory use:
  - g→$17, h→$18, i→$19, tmp1→$8, Astart→ starting memory address of A
- Assembly code not exactly:

```
lw   $8,Astart($19)        #tmp1←mem[Astart+i]
add  $17,$18,$8            #g←h+tmp1
```

Use: to access to vectors/arrays

- C code:
  - g=h+A[i]
- Registers and memory use:
  - g→$17, h→$18, i→$19, tmp1→$8, Astart→ starting memory address of A
- Assembly code:

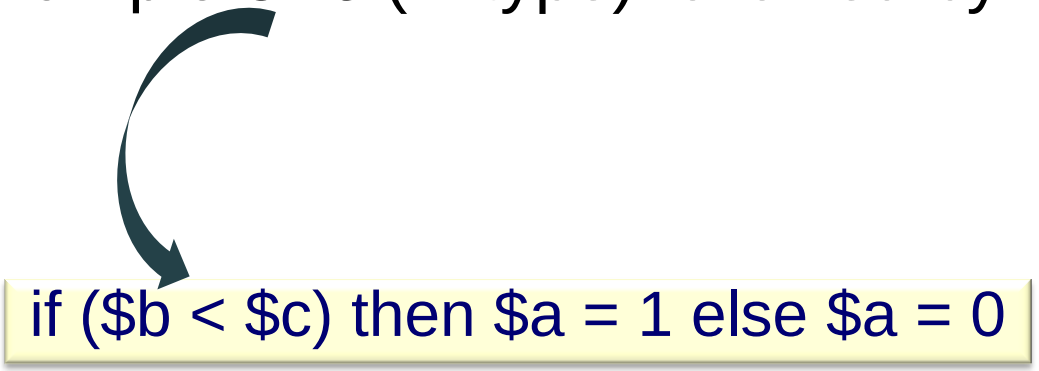
```
sll $9,$19,2          #tmp2←i*4
lw  $8,Astart($9)     #tmp1←mem[Astart+i]
add $17,$18,$8        #g←h+tmp1
```

sll: *Shift Left Logical*

What is this guy doing?

MIPS Conditional branch instructions

- Usually, I-type:
  - Compares content of two registers and depending on being equal jump: PC is loaded with sum of current PC and offset (`beq, bne`)
  - Other branch instructions determine to jump depending on conditions like “greater than” or “less than”, Example `slt` (R-type) followed by `bne`



Conditional branch instructions
Example beq

Opcode	rs	rt	immediate
000100	00111	00011	0000000000001111
Beq	Reg 7	Reg 3	offset=15

Format: I-type

Meaning: if(rs==rt) PC←PC+4+offset*4

Example: compares contents of registers 3 ($3) and 7 ($7). If equal, jumps to the instruction stored in memory location PC+4+15*4

Assembly code: `beq $3, $7, 60`

Notice, offset can be positive or negative: jump forward or backward.

Use of `slt` and `beq`: IF sentence

- C code:

```
if (a < b) then
    c = 100
else
    c = 500
```

- Register use: a→$16 b→$17, c→$18
- Assembly code:

Use of `slt` and `beq`: IF sentence

- C code:

  if (a < b) then

  c = 100

  else

  c = 500

- Register use: a→$16 b→$17, c→$18
- Assembly code:

```
            slt $18,$16,$17    # $18 = 1 if $16<$17
            beq $18,$0,ELSE    # if $18==0 jump ELSE
            addi $18,$0,100    # $18 = 100
            j CONTINUE         # jump to continue
ELSE :      addi $18,$0,500    # $18 = 500
CONTINUE:       ...
```

Can this be done with less, but the same type of instructions?

MIPS Unconditional branch instructions

- Usually, J-type:
  - Jump to a destination address determined by address operand and the 4 most significant bits of PC (`j, jal`)
  - Instruction `jal` stores content of PC in register 31 ($31), to know the way back to a calling procedure after finishing
  - Instruction `jr` jumps to the address stored in a register. Its format is R-type

Unconditional branch instructions
Example j

Opcode	address
000010	00000000000000000000001111
j	15

≡ << 2 ≡ ×4

Format: J-type

Meaning: PC←(PC+4)[31:28],address,00

Example: content PC becomes jump address determined by concatenating the 4 most significant bits of PC and using the 26 bits of the field address and two zeros

Assembly code: `j 60`

Unconditional branch instructions Example j

- PC= 12_{10}, machine instruction for `J 4000`?

  000010 00000000000000001111101000

  Opcode address= 1000_{10}

- If PC= 12_{10} can jump j reach memory address $3FF00118_{hex}$?

  12_{10} = **0000** 00000000000000000000001100$_2$

  3FF00118h = **0011** 11111111000000000100011000$_2$

  Infeasible: it is outside the accessible range of 2^{26} words for that value of PC (there are 2^4 segments)

Unconditional branch instructions Example jr

Opcode	rs				Funct
000000	00111	00000	00000	00000	001000
OpALU	Reg 7				jr

Format: R-type

Meaning: PC←rs

Example: loads PC with the value stored in register 7 ($7)

Assembly code: `jr $7`

Unconditional branch instructions
Example jal (*jump and link*)

Opcode	address
000011	00000000000000000000001111
jal	15

≡ << 2 ≡ ×4

Format: J-type

Meaning: $31←PC+4; PC←(PC+4)[31:28],address,00

Example: stores content of PC (+4) into register 31; where to proceed after finishing

content PC becomes jump address: concatenate 4 most significant bits of PC and paste the 26 bits of the field address and two zeros

Assembly code: `jal 60`

Translation of high-level structures to MIPS assembly language

ASSEMBLY LANGUAGE PROGRAMMING

Compiling IF statements

- C code

```
        if (i==j) goto L1
        f=g+h;
L1:     f=f-i;
```

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

- Assembly code

```
f       in $s3
g       in $s4
h       in $s5
i       in $s1
j       in $s2
```

Compiling IF statements

- C code

```
        if (i==j) goto L1
        f=g+h;
L1:     f=f-i;
```

- Assembly code

```
        beq $s1,$s2,L1
        add $s3,$s4,$s5
L1:     sub $s3,$s3,$s1
```

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

```
        bne $s1,$s2,ELS
        add $s3,$s4,$s5
        j END
ELS:    sub $s3,$s4,$s5
END:    ...
```

Compiling REPEAT-UNTIL loops

- C code

Repeat

    g=g+A[i];

    i=i+j;

Until (i==h)

- Assembly code

g in $17

i in $18

j in $19

h in $20

tmp1 in $8

tmp2 in $9

Astart →starting memory address of A

Compiling REPEAT-UNTIL loops

- C code

Repeat
    g=g+A[i];
    i=i+j;
Until (i==h)

g in $17
i in $18
j in $19
h in $20
tmp1 in $8
tmp2 in $9
Astart → starting memory address of A

- Assembly code

```
RU: sll  $8,$18,2   #tmp1←i*4
    lw   $9,Astart($8)
                    #tmp2←mem[Astart+tmp1]
    add  $17,$17,$9 #g←g+tmp2
    add  $18,$18,$19 #i←i+j
    bne  $18,$20,RU
```

sll: *Shift Left Logical*

Compiling WHILE loops

- C code

```
while (A[i]==k)
    i=i+j;
```

i in $19
j in $20
k in $21
tmp1 in $8
tmp2 in $9
Astart → starting memory address of A

- Assembly code

Compiling WHILE loops

- C code

```
while (A[i]==k)
    i=i+j;
```

i in $19
j in $20
k in $21
tmp1 in $8
tmp2 in $9
Astart → starting memory address of A

- Assembly code

```
WH:sll   $8,$19,2   #tmp1←i*4
   lw    $9,Astart($8)
                    #tmp2←mem[Astart+tmp1]
   bne   $9, $21, Exit
   add   $19,$19,$20 #i←i+j
   j     WH
Exit:…
```

sll: *Shift Left Logical*.

Compiling CASE-SWITCH statements

- C code

```
switch (k) {
  case 0: f=i+j; break;
  case 1: f=g+h; break;
  case 2: f=g-h; break;
  case 3: f=i-j; break;
}
```

```
f    in $16
g    in $17
h    in $18
i    in $19
j    in $20
k    in $21
tmp1 in $8
tmp2 in $9
```

JumpTable 0	dir L0
4	dir L1
8	dir L2
12	dir L3

- Assembly code

Compiling CASE-SWITCH statements

- C code

```
switch (k) {
  case 0: f=i+j; break;
  case 1: f=g+h; break;
  case 2: f=g-h; break;
  case 3: f=i-j; break;
}
```

JumpTable 0	dir L0
4	dir L1
8	dir L2
12	dir L3

```
f    in $16
g    in $17
h    in $18
i    in $19
j    in $20
k    in $21
tmp1 in $8
tmp2 in $9
```

- Assembly code

```
 L0: add   $16,$19,$20 #f←i+j
     j     exit
 L1: add   $16,$17,$18 #f←g+h
     j     exit
 L2: sub   $16,$17,$18 #f←g-h
     j     exit
 L3: sub   $16,$19,$20 #f←i-j
     j     exit
exit: …
```

Compiling CASE-SWITCH statements

- C code

```
switch (k) {
  case 0: f=i+j; break;
  case 1: f=g+h; break;
  case 2: f=g-h; break;
  case 3: f=i-j; break;
}
```

JumpTable	0	dir L0
	4	dir L1
	8	dir L2
	12	dir L3

f in $16
g in $17
h in $18
i in $19
j in $20
k in $21
tmp1 in $8
tmp2 in $9

- Assembly code

```
        sll   $9,$21,2     #tmp2←k*4
        lw    $8,JumpTable($9)
              #tmp1←mem[JumpTable+tmp2]
        jr    $8      #jump to JumpTable(k)
  L0:   add   $16,$19,$20 #f←i+j
        j     exit
  L1:   add   $16,$17,$18 #f←g+h
        j     exit
  L2:   sub   $16,$17,$18 #f←g-h
        j     exit
  L3:   sub   $16,$19,$20 #f←i-j
        j     exit
exit:   …
```

Procedure calls

- The instruction set facilitates
  - calling procedures: using `jal` instruction (jump and link). The return address is stored in register $31 ($ra)

```
jal ProcedureAddress
```

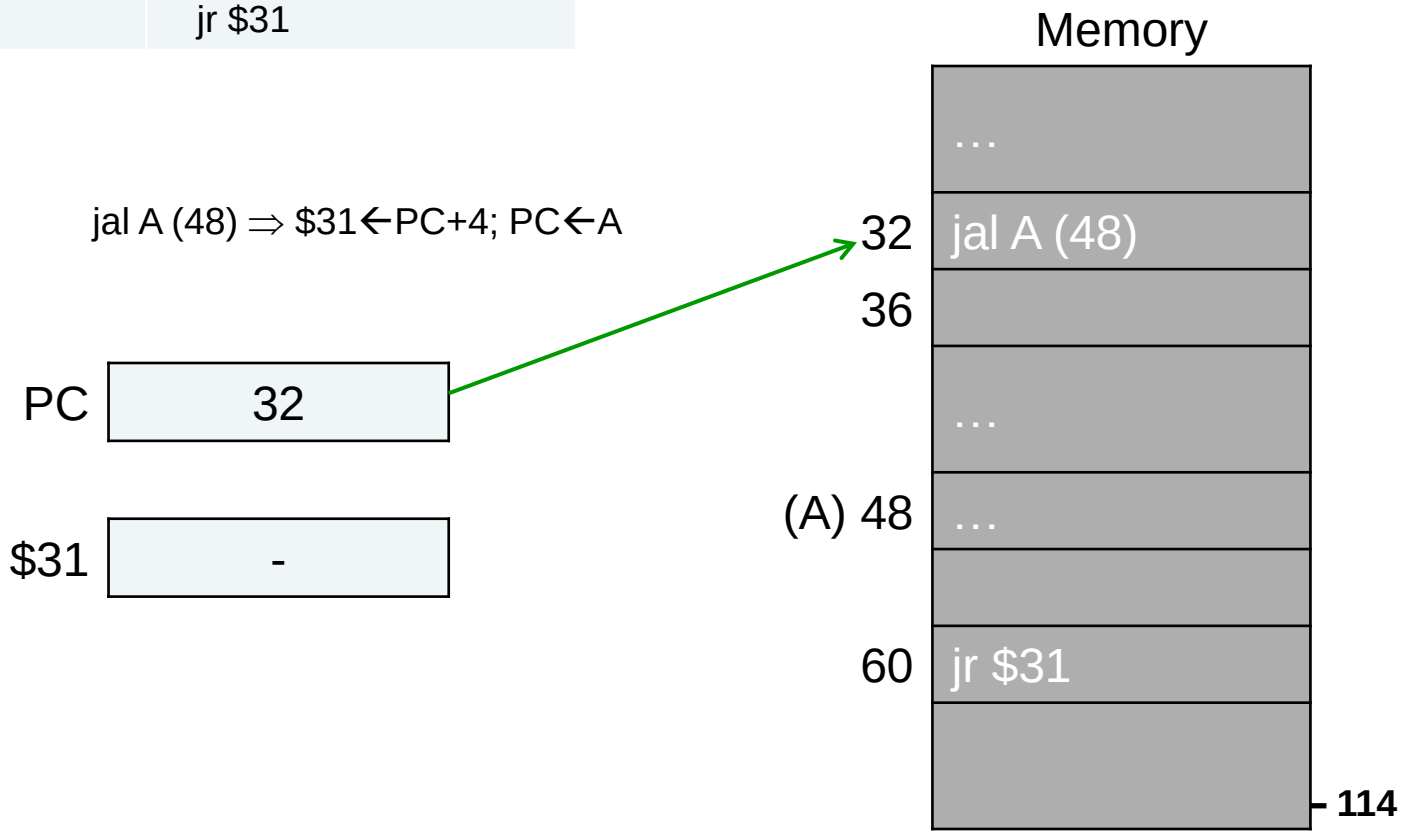
  - returning to the next instruction of the caller procedure after finishing called procedure via the `jr` instruction (jump to register)

```
jr $31
```

Procedure calls example

- Calling procedure A and returning

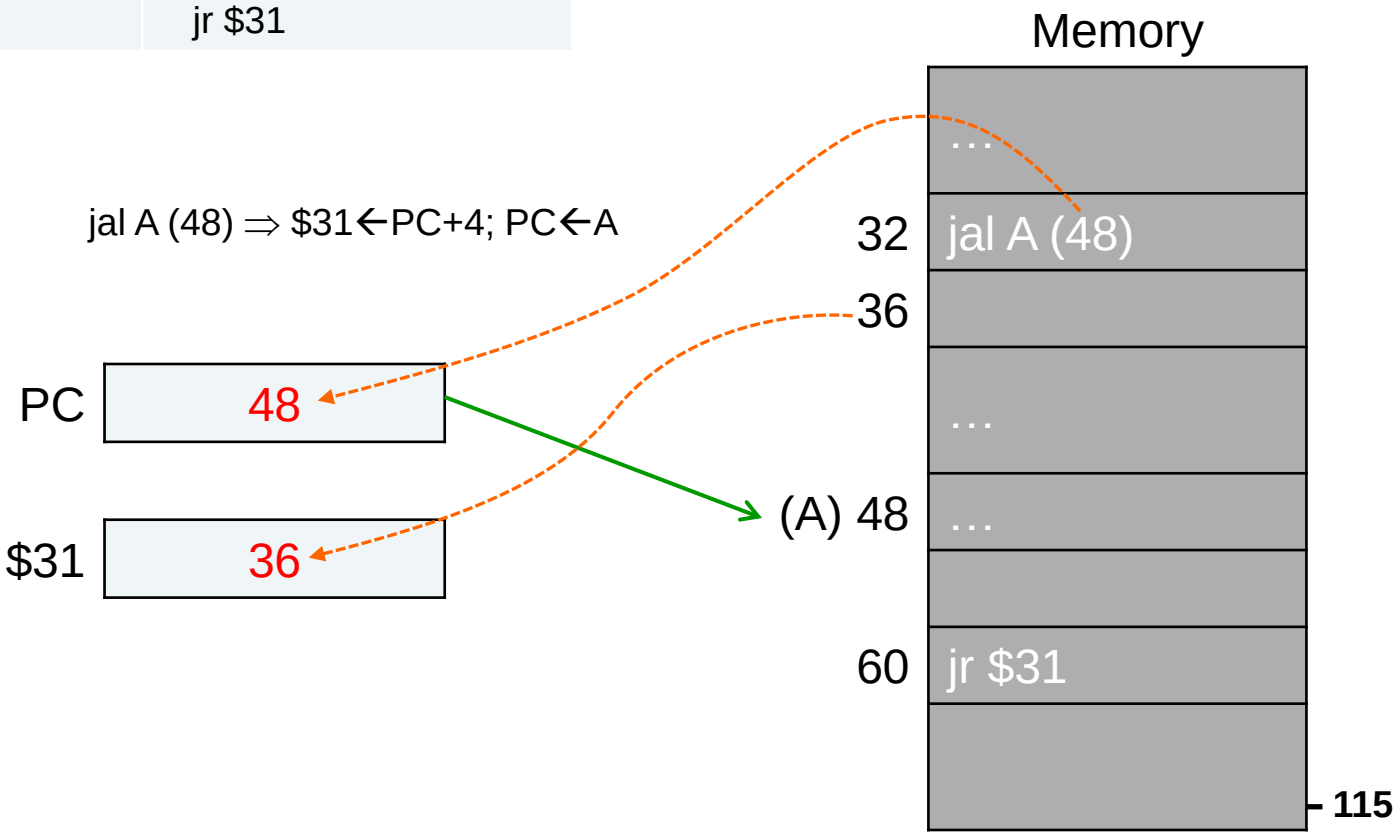
Main block	Basic block A
… jal A …	A: … … jr $31



Procedure calls (II): example

- Calling procedure A and returning

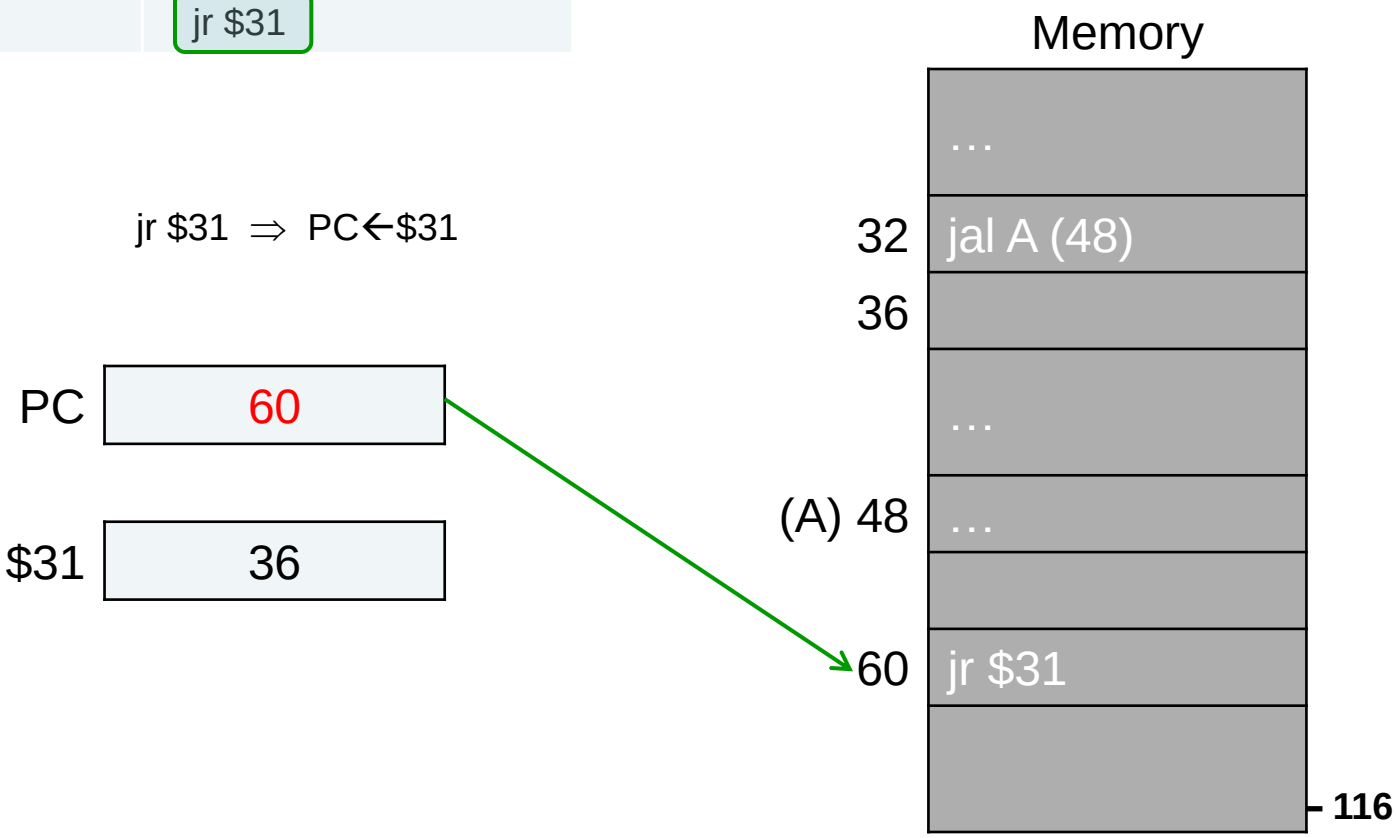
Main block	Basic block A
… jal A …	A: … … jr $31



Procedure calls (II): example

- Calling procedure A and returning

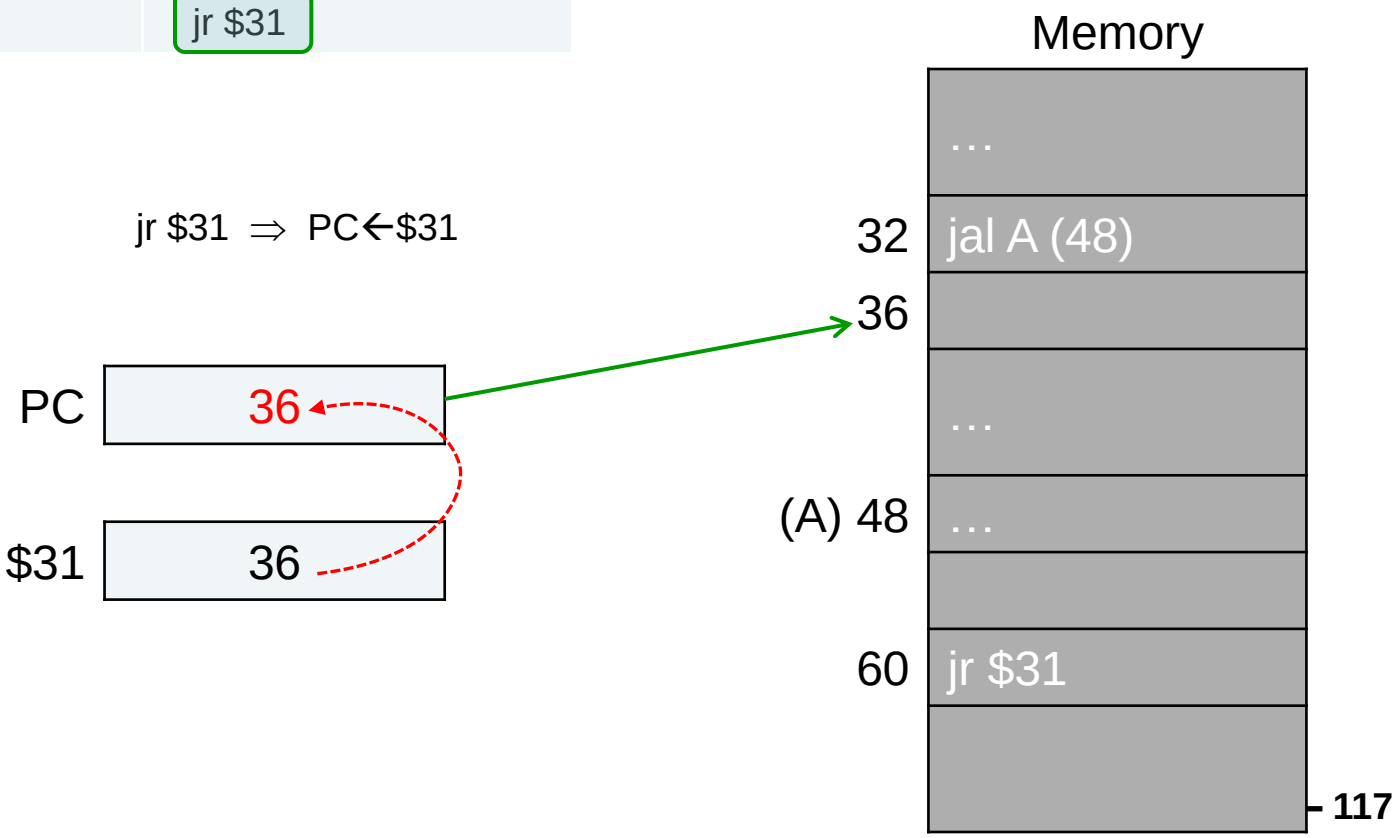
Main block	Basic block A
… jal A …	A: … … jr $31



Procedure calls (II): example

- Calling procedure A and returning

Main block	Basic block A
… jal A …	A: … … jr $31



Procedure calls convention

- Challenge of nested procedures
  - Store arguments of the procedures in stack

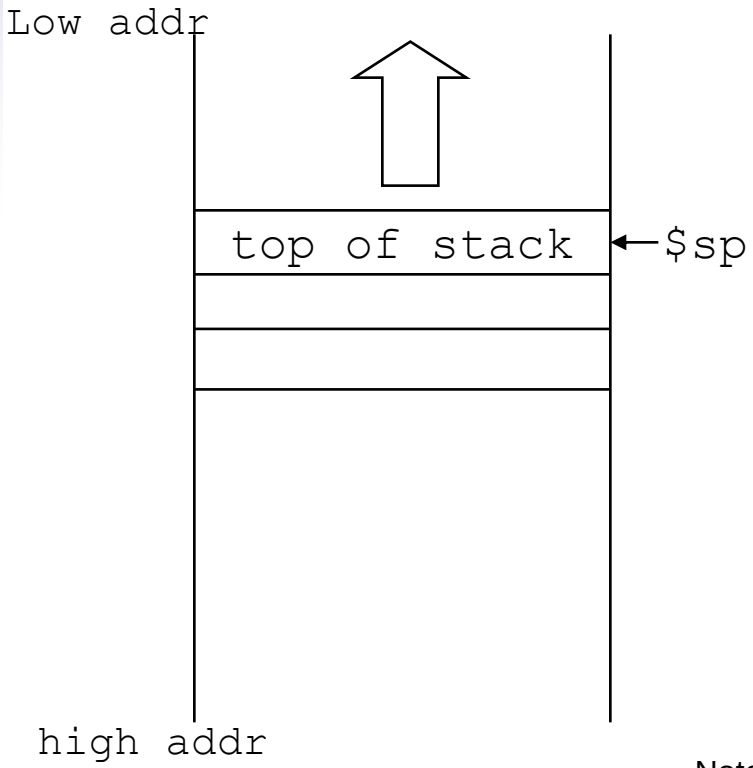
Sequential procedure calls (`jal`) overwrite register $31

STACK (LIFO): 1st return address is introduced by the last procedure call

STACK POINTER (SP)

The stack, LIFO storage

- A reserved area of memory for temporary storage with “Last In, First Out” structure
- $sp ($29) is used to address the stack. Grows bottom up from high addresses to low addresses



- Stack grows, $sp gets smaller
- stack operations:
  - PUSH: to add data onto the stack
    - $sp= $sp-4 (make space)
    - data stored on stack at new $sp
  - POP: remove data from stack
    - take data from stack at $sp
    - $sp= $sp+4 (release space)

Note: this is a NZ variant of the book

Repeat: MIPS Register convention

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	no
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

Procedure call conventions

- **Nested procedure calls:**
  - Reg. $29 ($sp): stores the last value pushed in the stack (SP, stack pointer)
- **Passing arguments (parameters):**
  - Arguments of called procedure are passed by registers $4-$7($a0 - $a3), if there are more than 4 arguments, they are passed on the stack
  - The results are placed in registers $2-$3 ($v0-$v1)
  - The local variables of the nested procedures are pushed onto the stack too

Six Steps in Execution of a Procedure

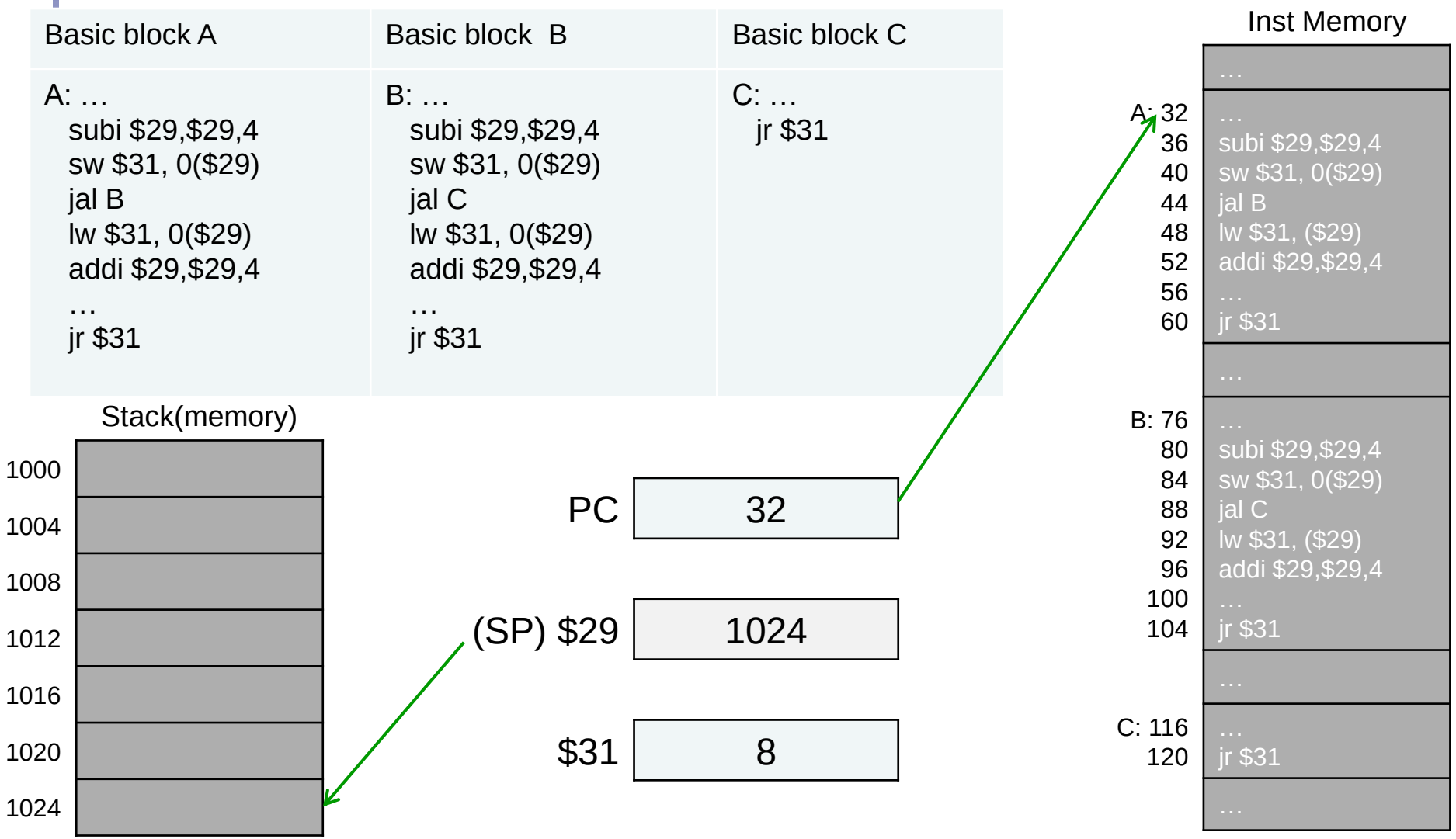
1. Main routine (caller) places parameters in a place where the procedure (callee) can access them
   - `$a0` - `$a3`: four argument registers
2. Caller transfers control to the callee
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where the caller can access it
   - `$v0` - `$v1`: two value registers for result values
6. Callee returns control to the caller
   - `$ra`: one return address register to return to place of call (the point of origin, register 31 in this design)

Procedure calls: Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

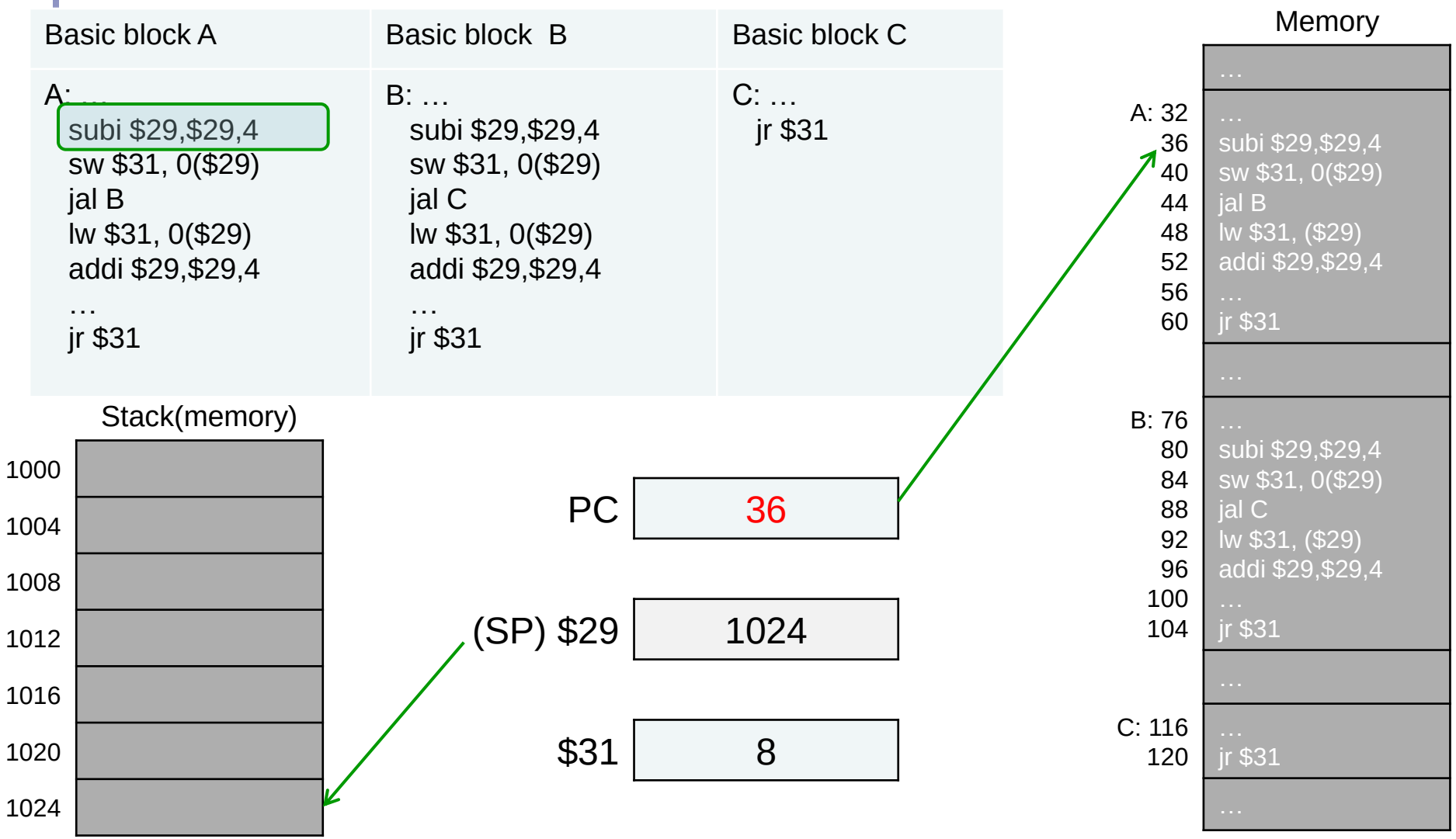


Procedure calls: Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

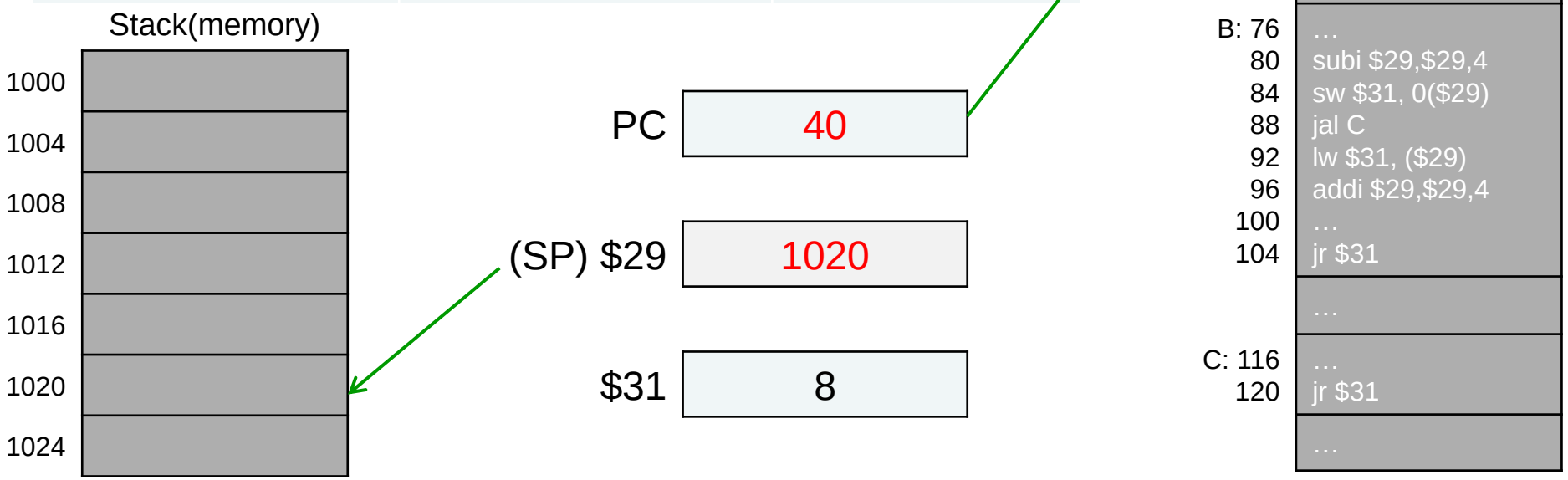


Procedure calls: Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

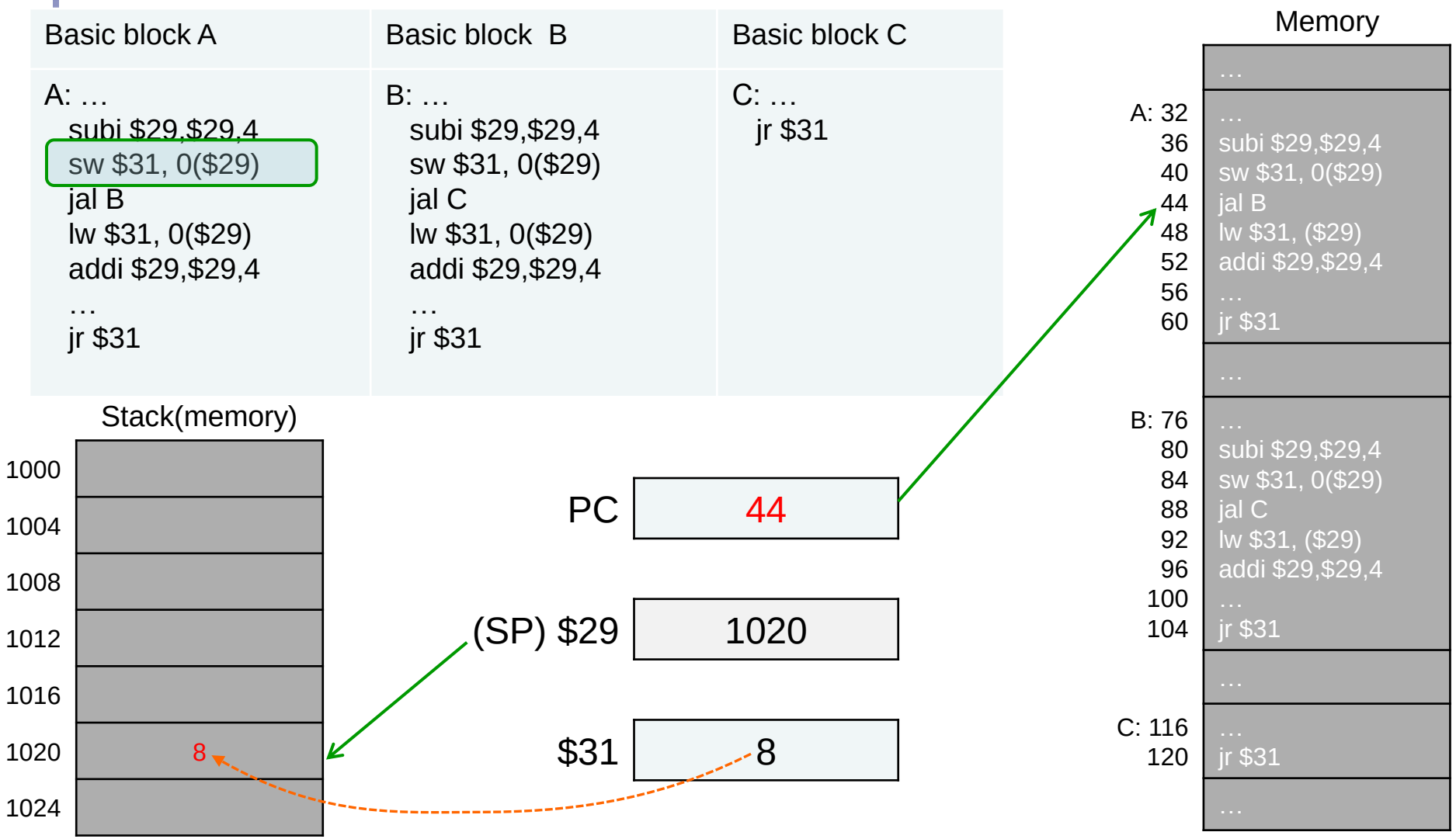


Procedure calls: Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

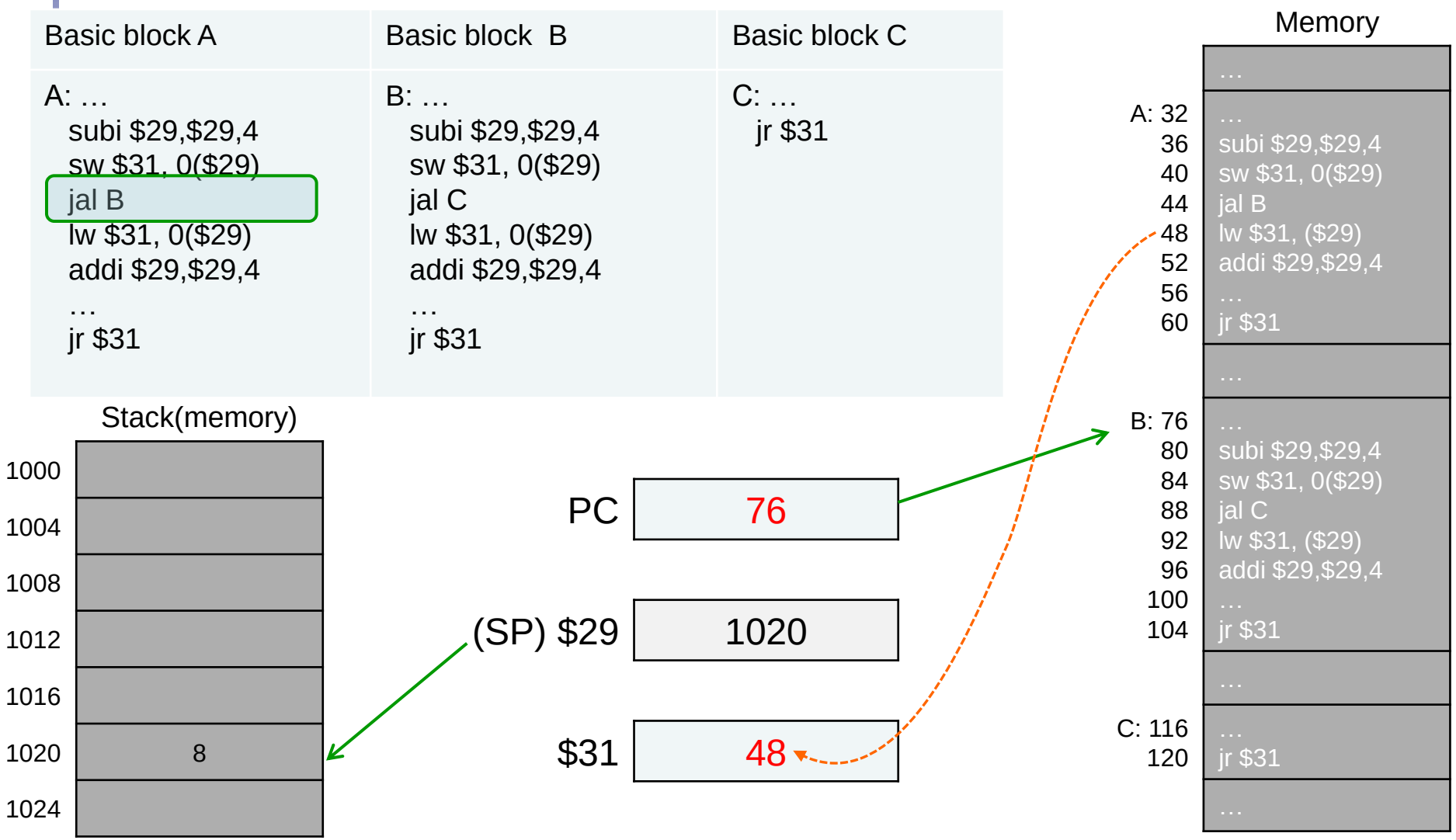


Procedure calls: Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

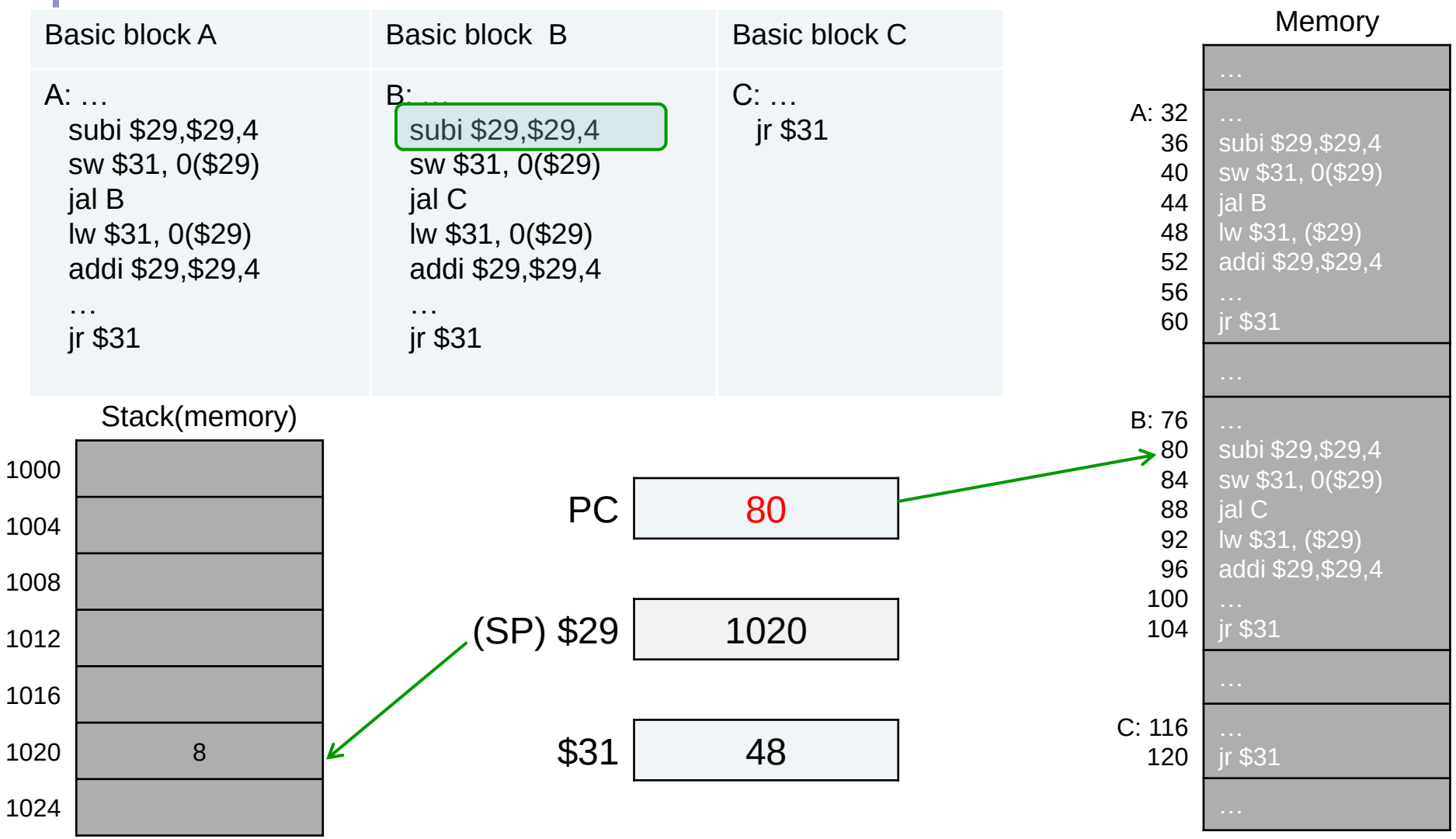


Procedure calls: Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

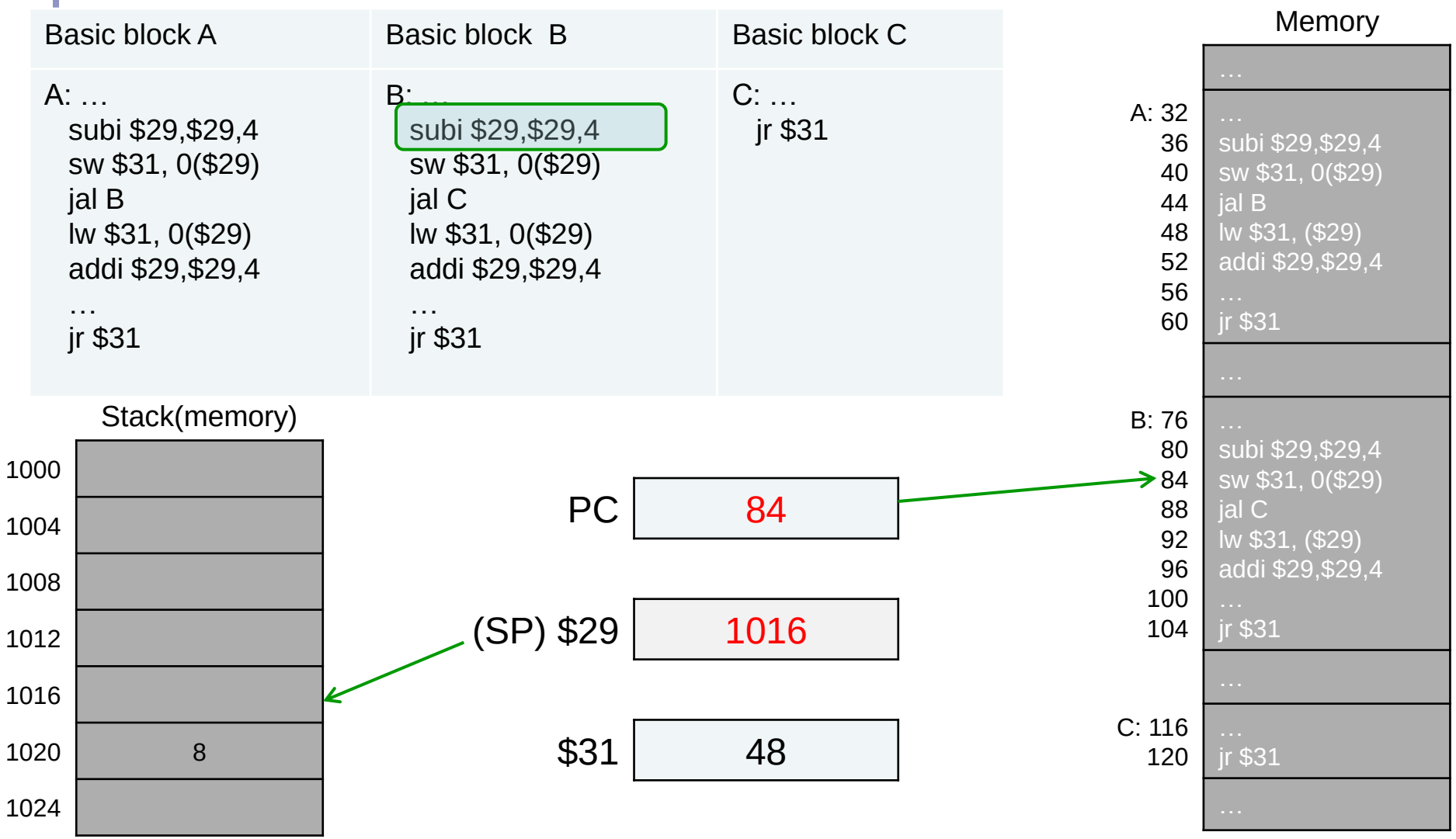


Procedure calls: Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

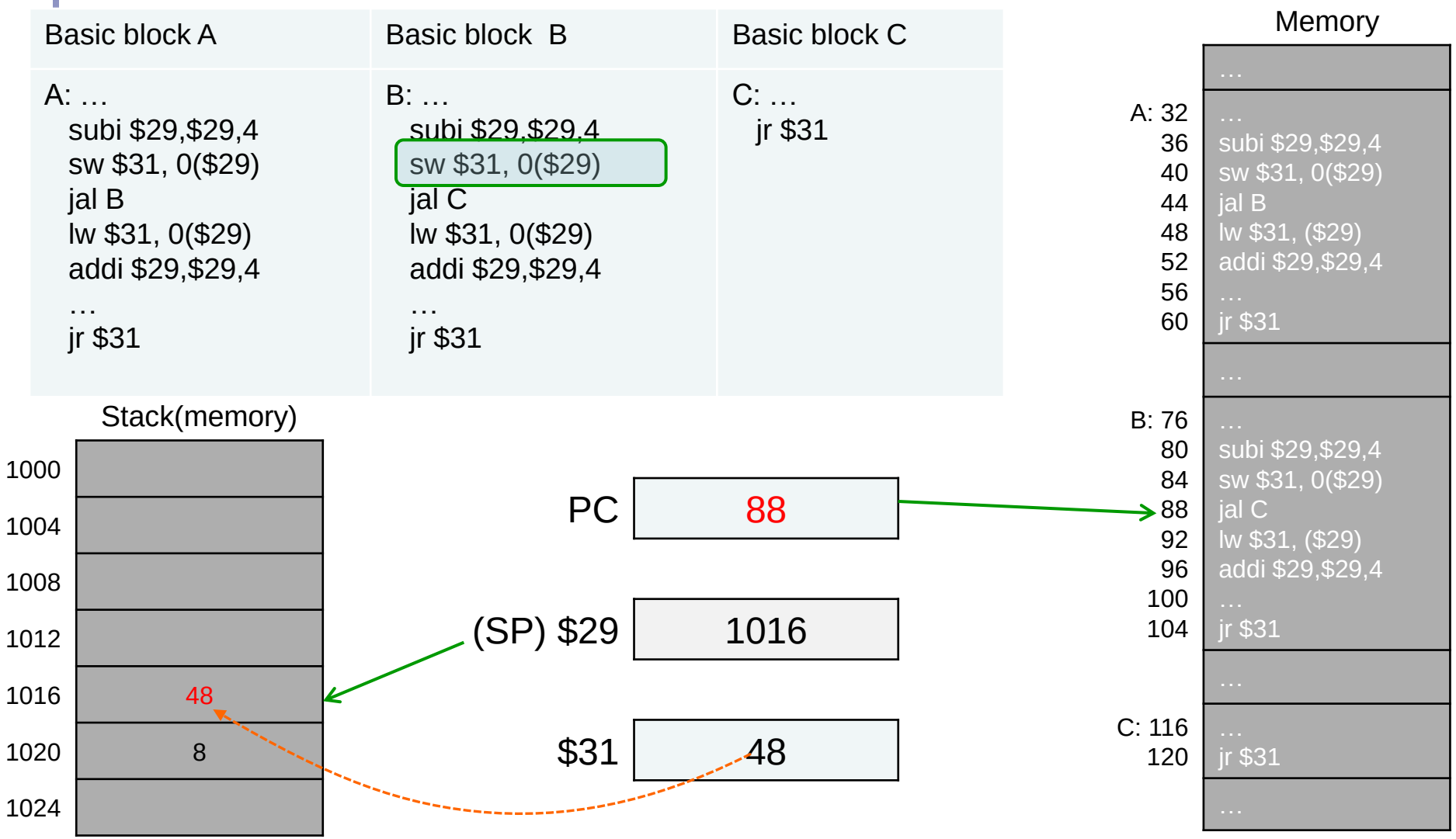


Procedure calls: Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

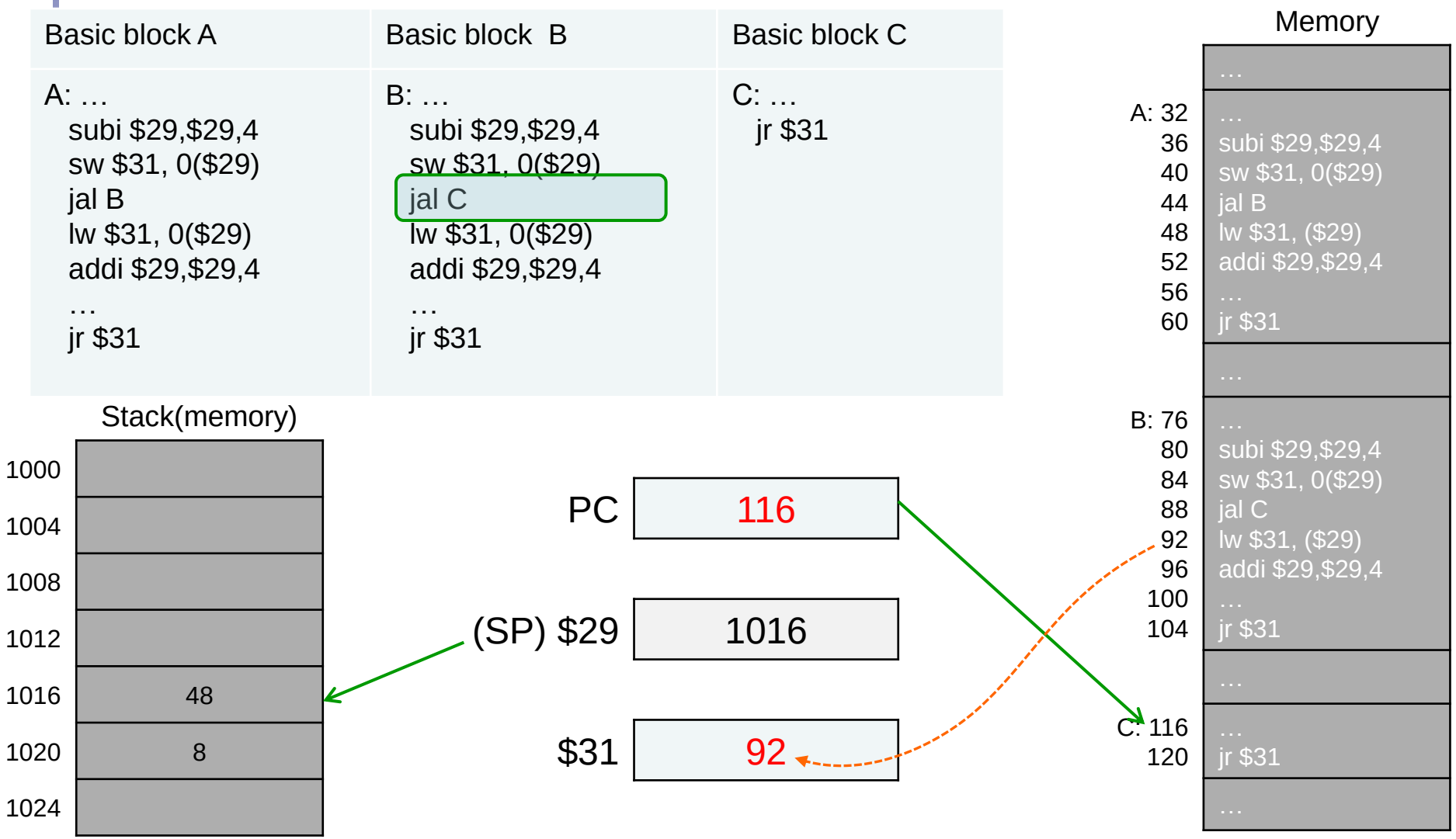


Procedure calls: Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

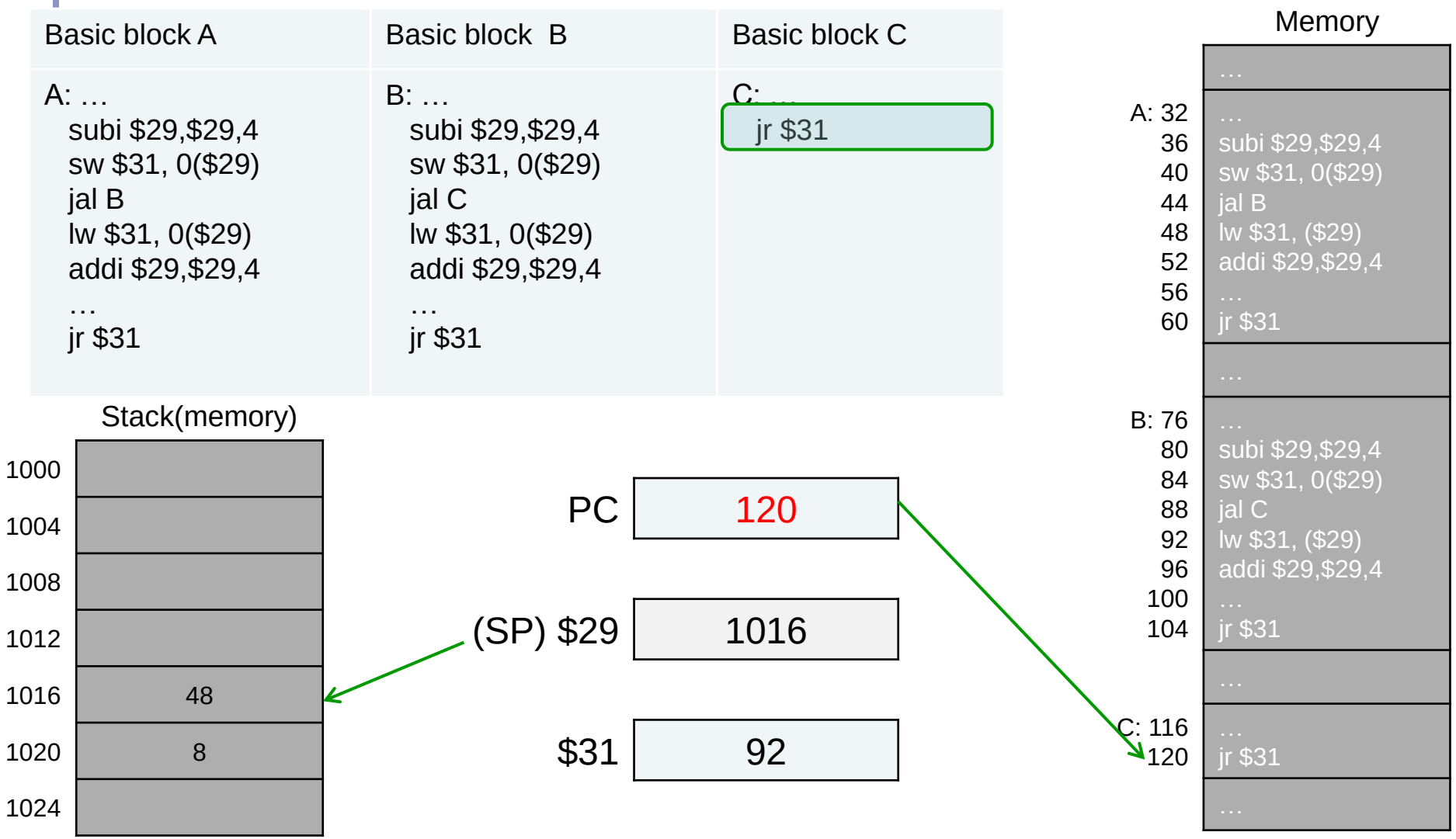


Procedure calls: Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

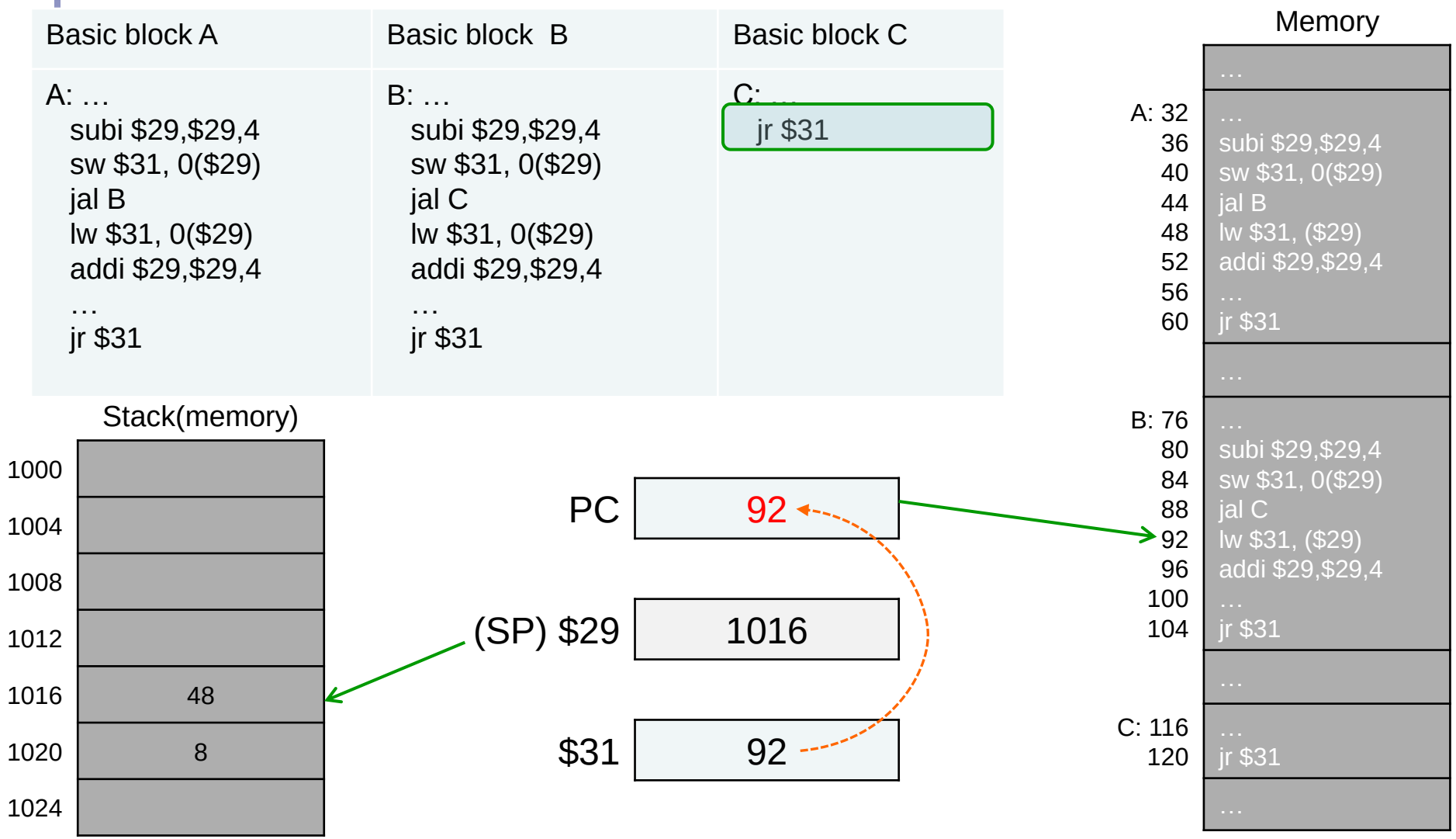


Procedure calls: Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

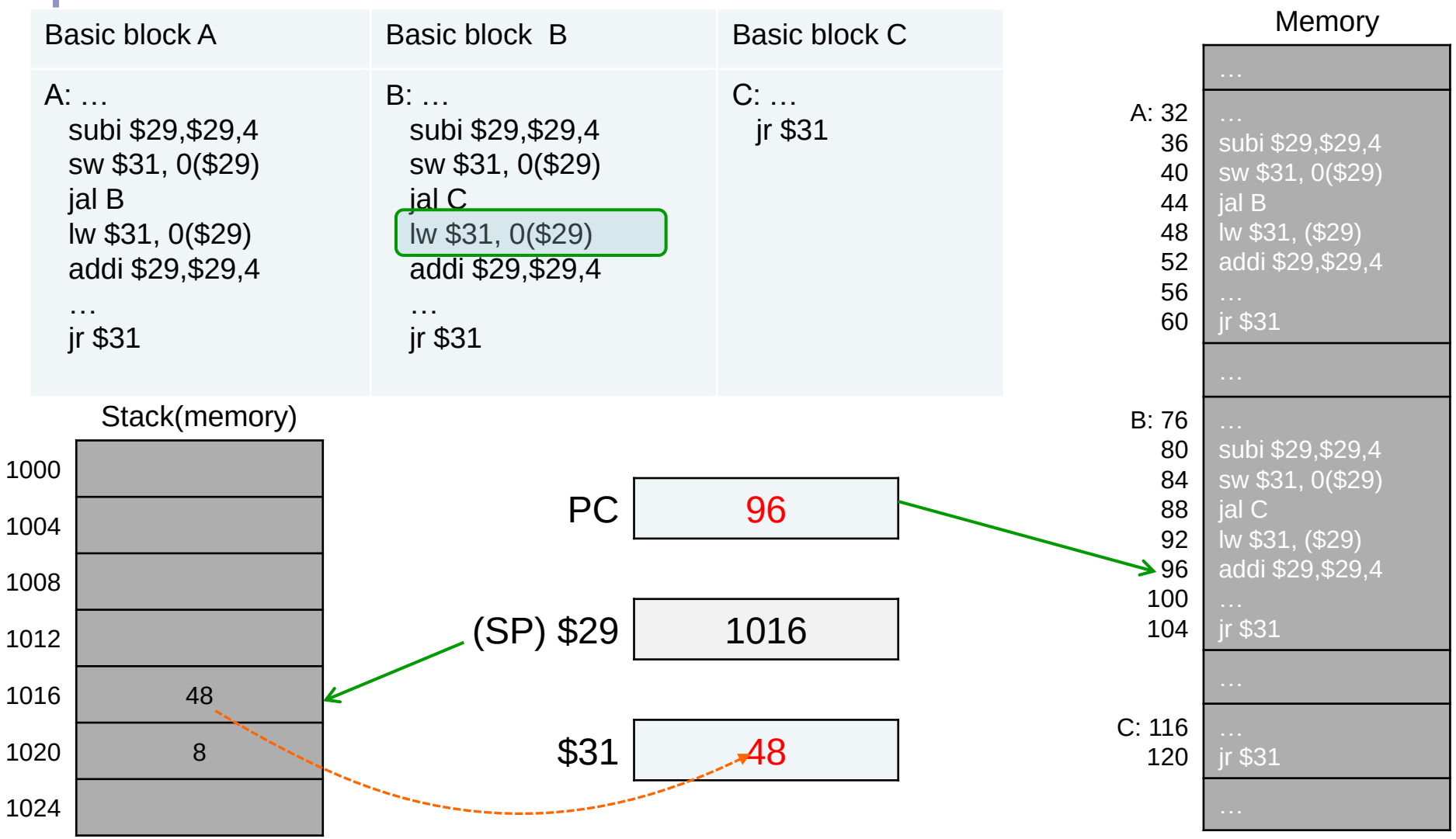


Procedure calls: Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

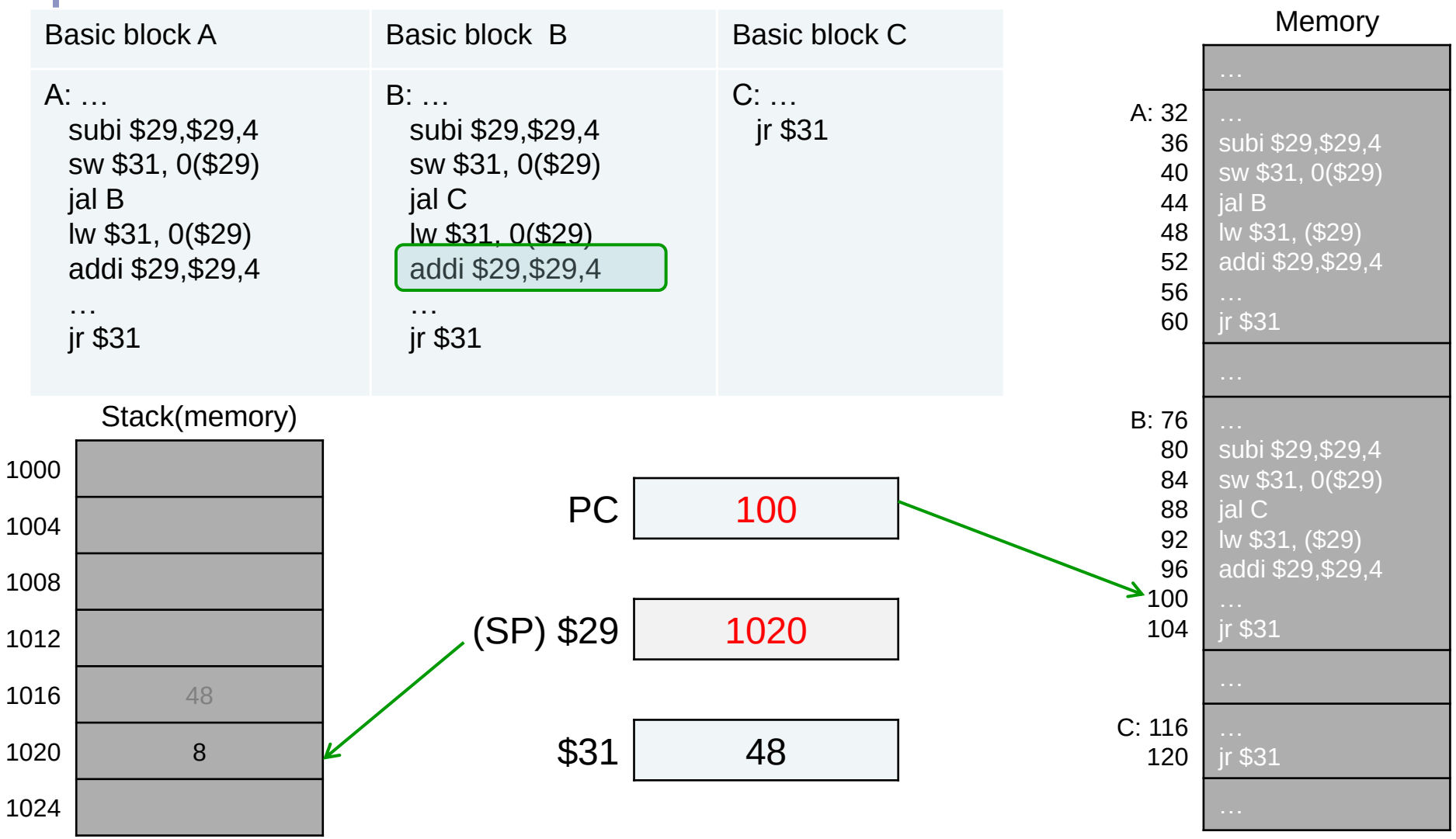


Procedure calls : Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

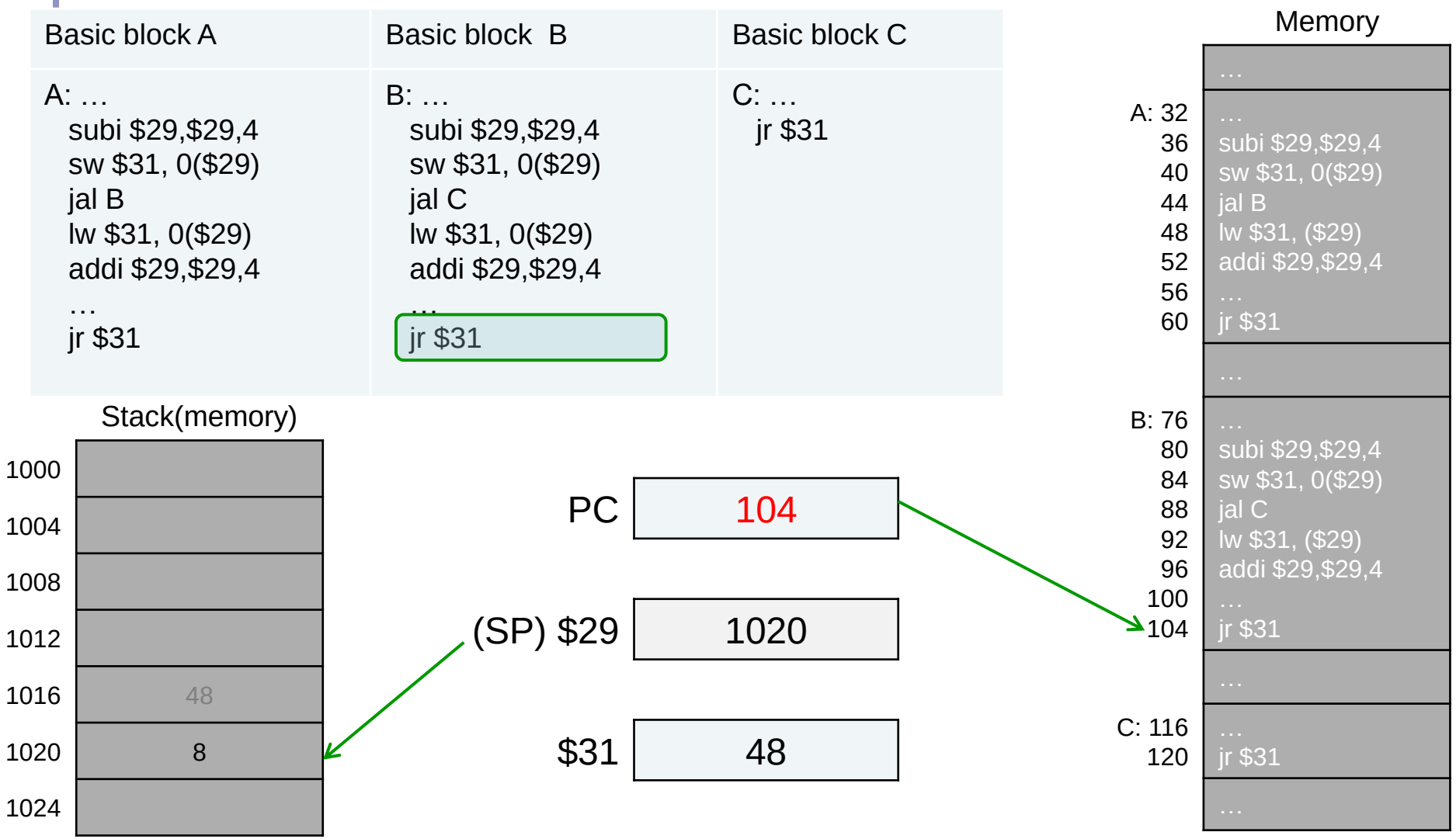


Procedure calls : Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

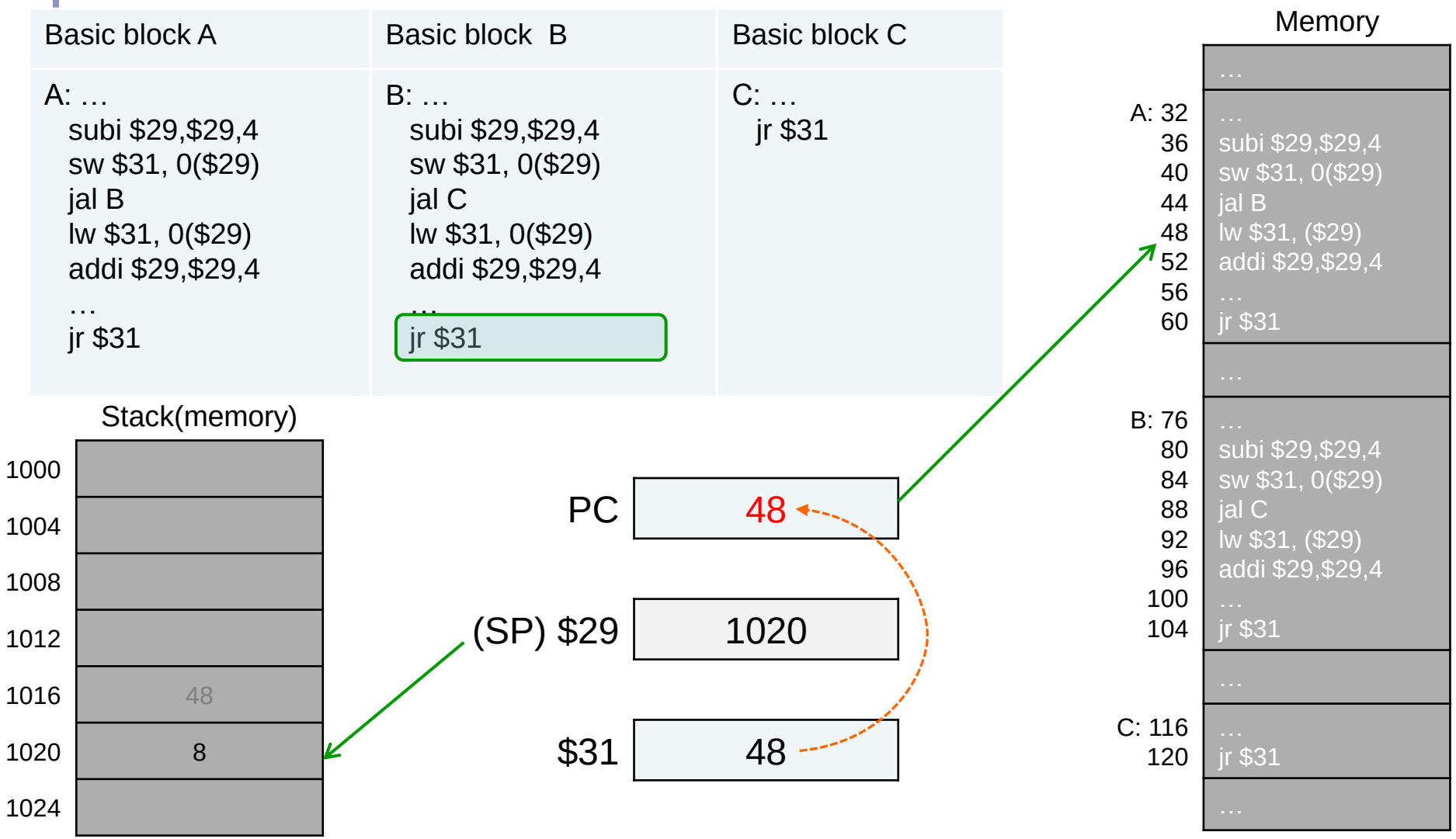


Procedure calls : Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

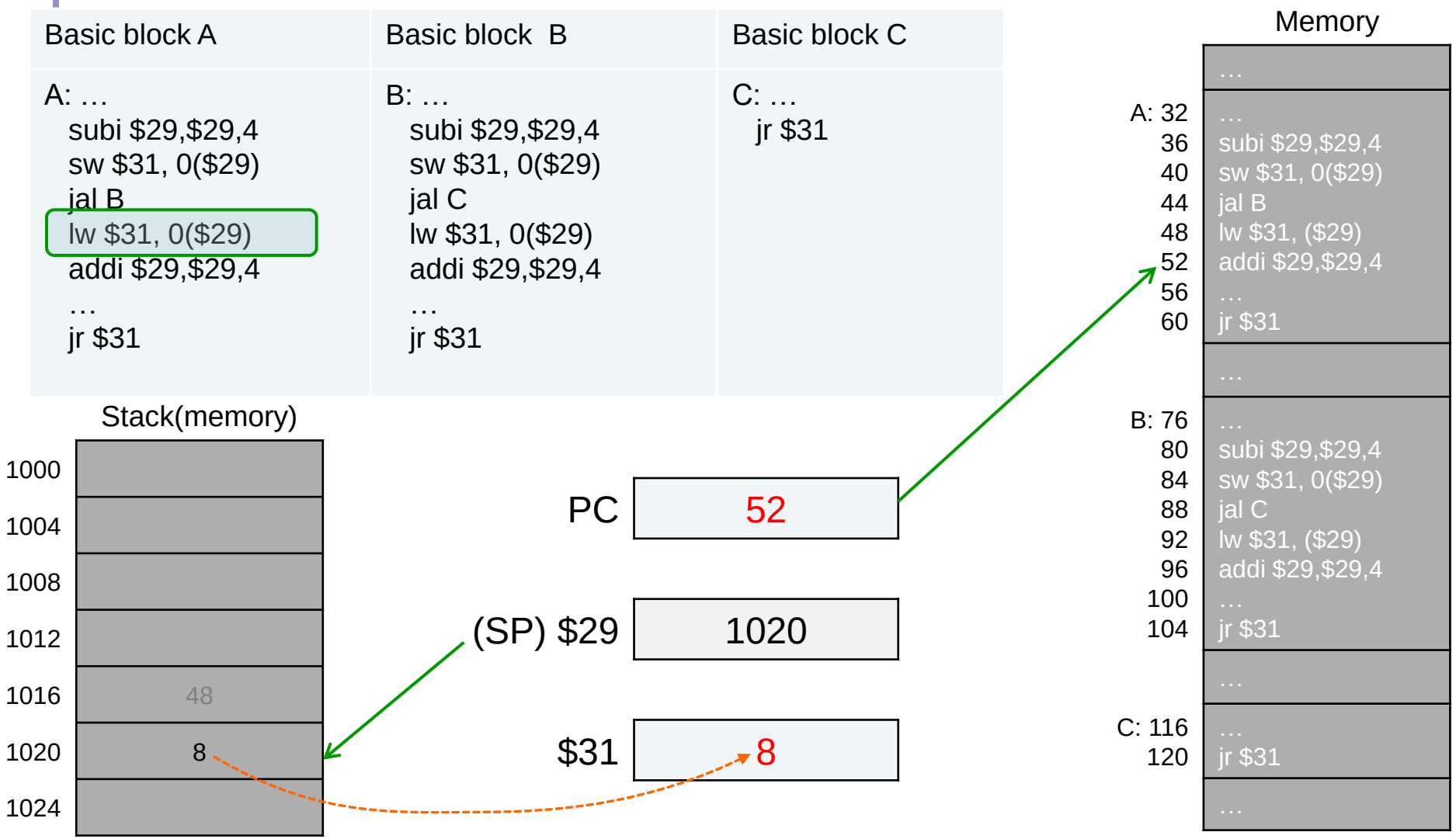


Procedure calls : Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

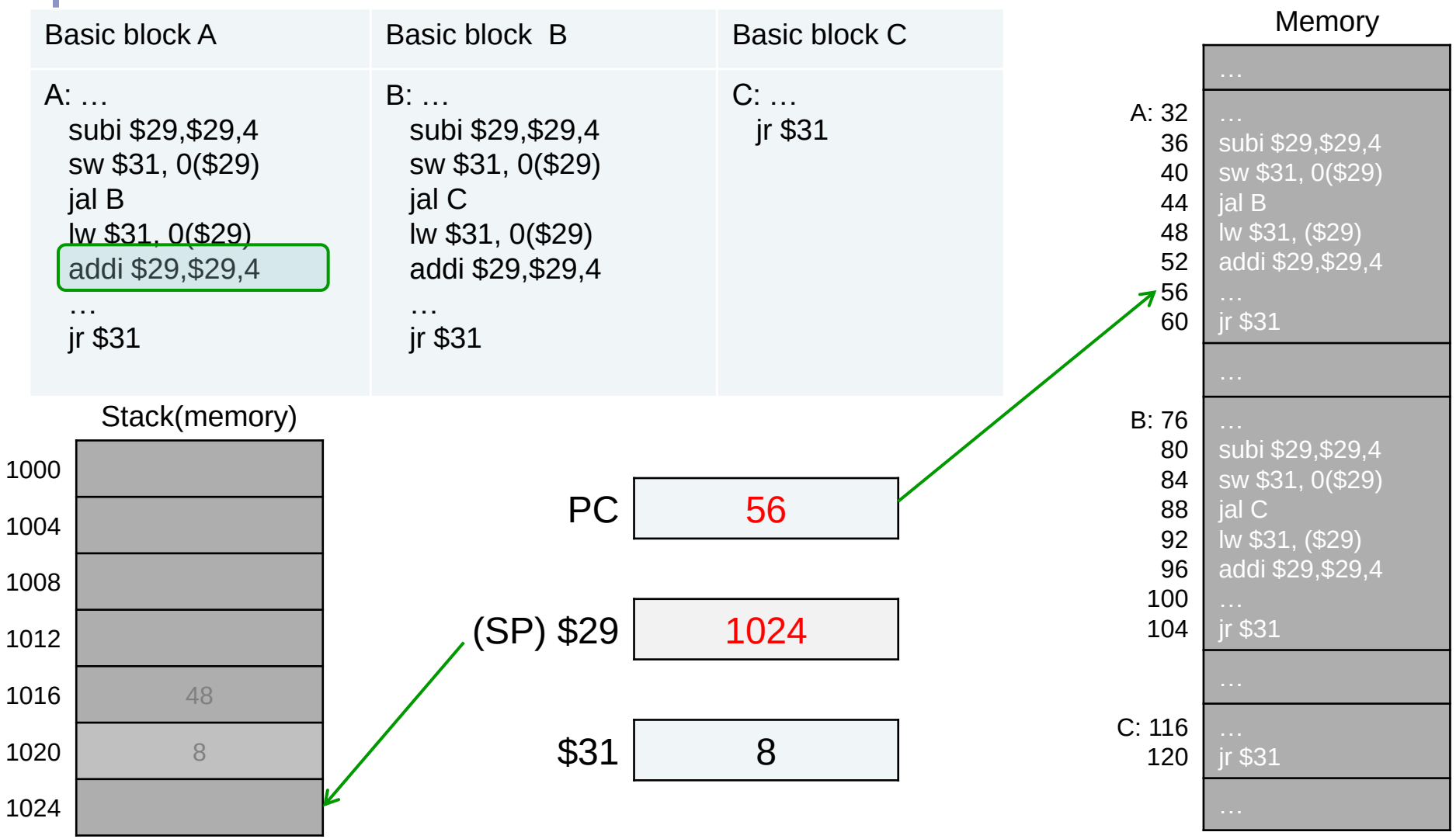


Procedure calls : Nested procedures

- Calling 3 nested proc.: A→B→C

Basic block A	Basic block B	Basic block C

A: …
subi $29,$29,4
sw $31, 0($29)
jal B
lw $31, 0($29)
addi $29,$29,4
…
jr $31
	B: …
	subi $29,$29,4
	sw $31, 0($29)
	jal C
	lw $31, 0($29)
	addi $29,$29,4
	…
	jr $31
		C: …
		jr $31

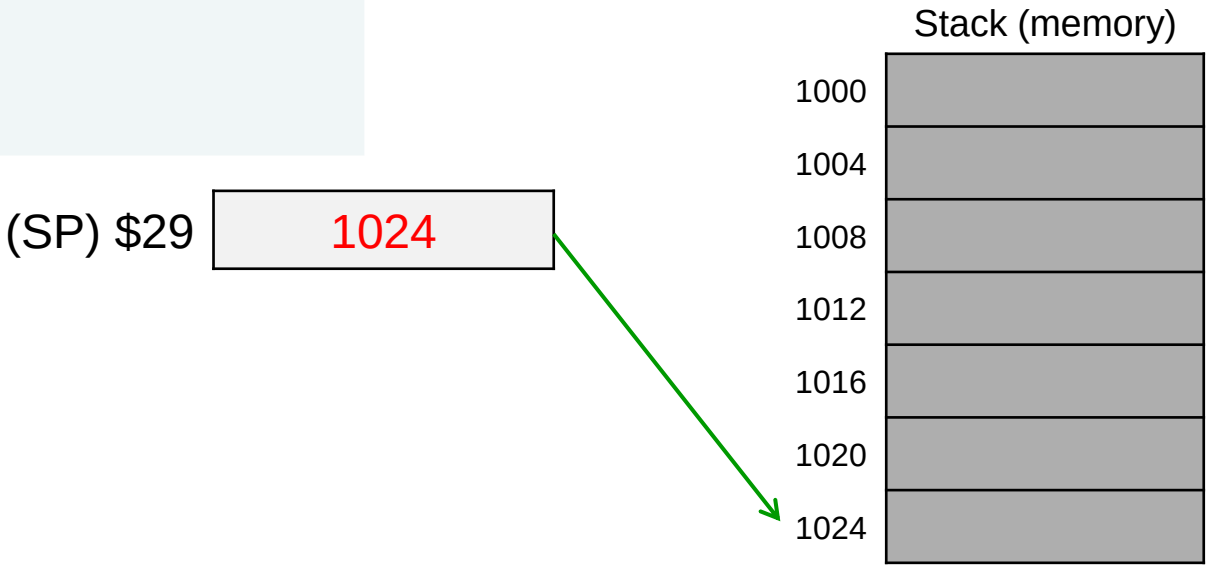


Procedure calls: Arguments

- Before calling a procedure, push parameters ($8,$9,$10) onto the stack (adjusting the stack pointer $29)

```
…
subi $29,$29,4
sw $8, 0($29)
subi $29,$29,4
sw $9, 0($29)
subi $29,$29,4
sw $10, 0($29)
jal B
…
```

```
B: lw $4, 0($29)
   lw $5, 4($29)
   lw $6, 8($29)
   …
   jr $31
```

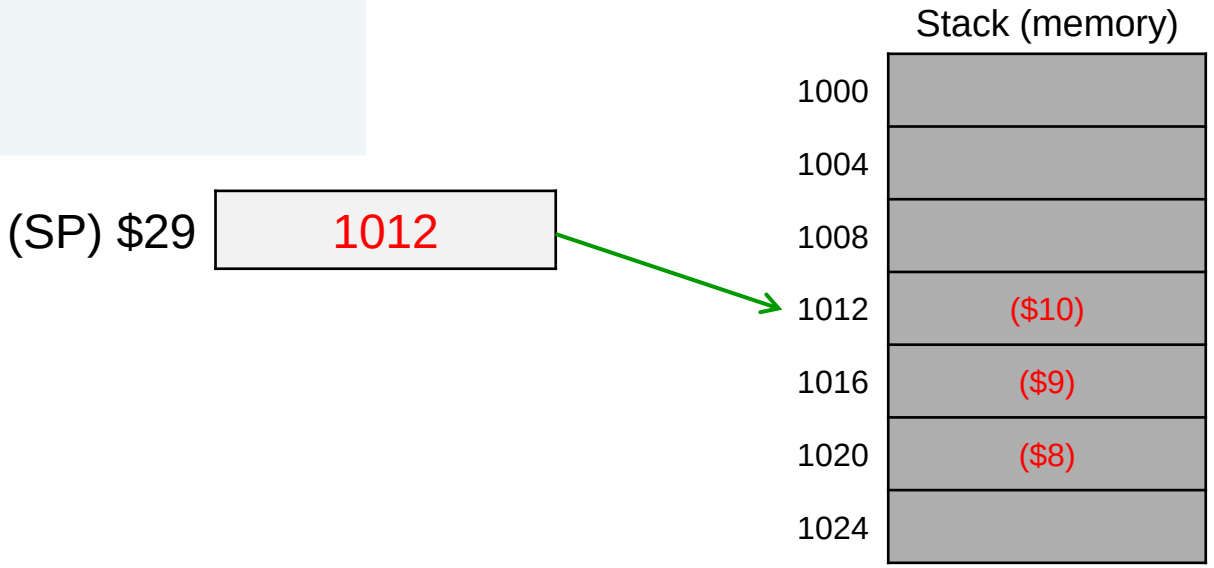


Procedure calls: Arguments

- Inside procedure B, the parameters are accessed from the stack pointer ($29) and different shifts (0,4,8)

```
…
subi $29,$29,4
sw $8, 0($29)
subi $29,$29,4
sw $9, 0($29)
subi $29,$29,4
sw $10, 0($29)
jal B
…
```

```
B: lw $4, 0($29)
   lw $5, 4($29)
   lw $6, 8($29)
   …
   jr $31
```

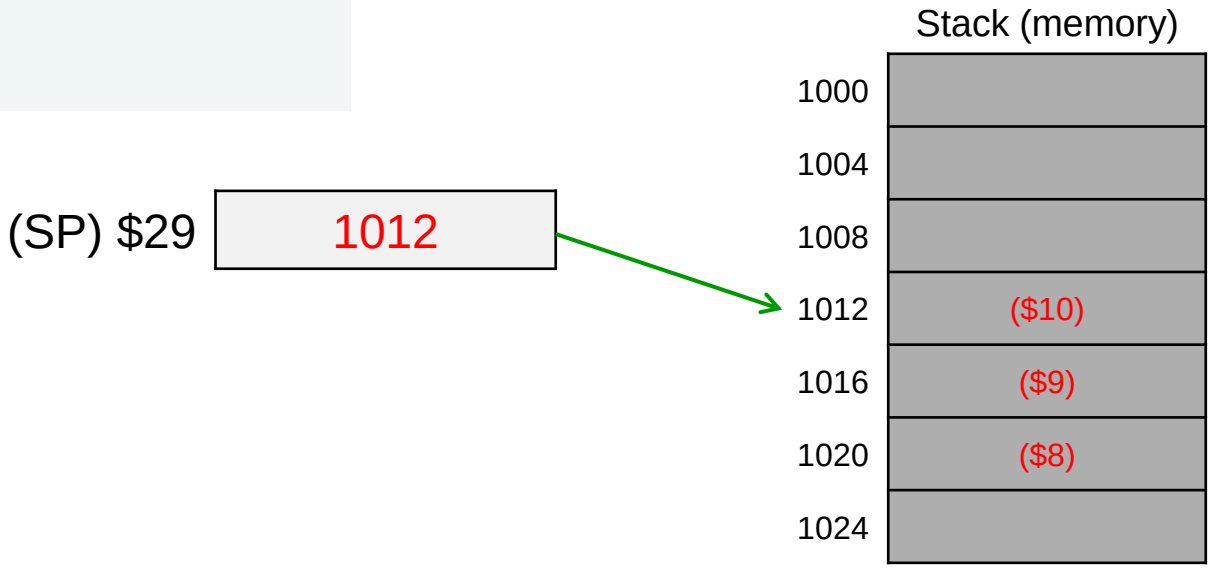


Procedure calls: Arguments

- Inside procedure B, the parameters are accessed from the stack pointer ($29) and different shifts (0,4,8)

```
…
subi $29,$29,12
sw $8, 8($29)
sw $9, 4($29)
sw $10, 0($29)
jal B
addi $29,$29,12
…
```

```
B: lw $4, 0($29)
   lw $5, 4($29)
   lw $6, 8($29)
   …
   jr $31
```



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

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
```

Bibliography

- D.A. PATTERSON and J.L. HENNESSY (2009). Computer organization and design: The hardware / software interface. 4th edition. Morgan Kaufmann
  - Chapters 1 and 2.
- P.M. ANASAGASTI (1998). Fundamentos de los computadores. 8th edition. Madrid. Ed. Thomson
  - Chapters 1 and 6.
- G. BANDERA, F.J. CORBERA et al. (2000). Tecnología de computadores. 1st edition. Málaga. Servicio de publicaciones de la UMA.
  - Chapter 2.