Assembly Language

Prof. Dr. Antônio Augusto Fröhlich
guto@lisha.ufsc.br
http://www.lisha.ufsc.br/~guto

Sep 2006

Assembly Language

"Assembly language, commonly called assembly, asm or symbolic machine code, is a human-readable notation for the machine language that a specific computer architecture uses. Machine language, a pattern of bits encoding machine operations, is made readable by replacing the raw values with symbols called mnemonics."

(Wikipedia)

Assembler

"An assembler creates object code by translating assembly instruction mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities."

(David Salomon)

Memory Access

- Program instructions and data are stored in memory
  - Must be brought into the CPU during execution
- Program instructions are automatically fetched by the CPU
  - Control-flow instructions guide the execution flow
- Data operands for ALU instructions are fetched by explicit instructions in the program
  - Data access instructions

CPU Registers

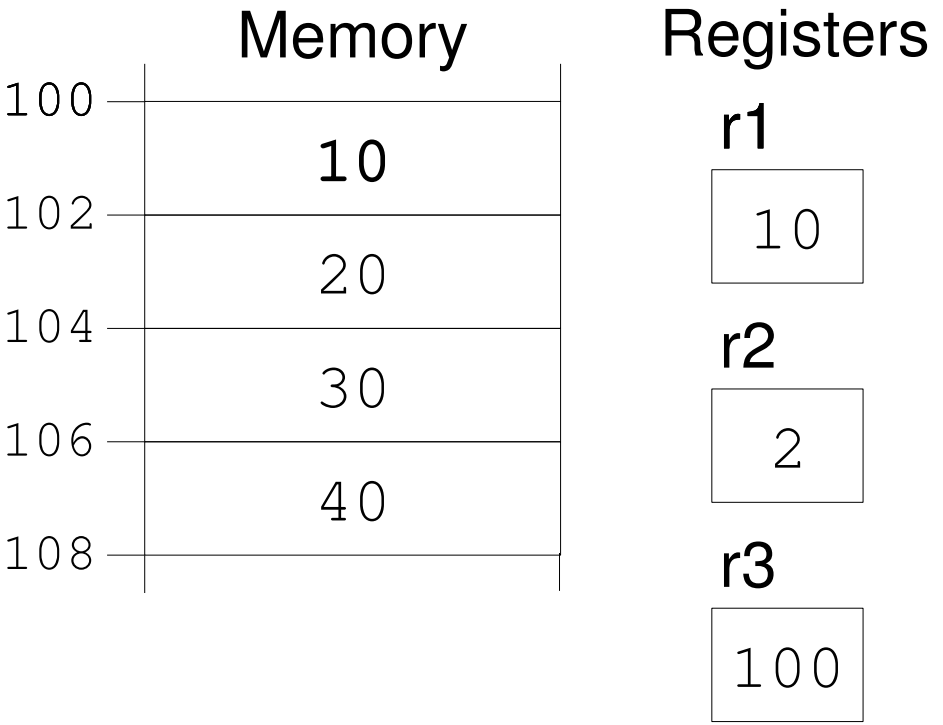
- General Purpose Registers (GPR)
  - Scratch storage
  - Data
  - Addresses
  - Constants (0/1/-1)
  - Indexes for data structures
  - Base/segment to segment memory
- Floating Point Registers (FPR)
  - Floating point data

CPU Registers

- Control registers
  - Program Counter (PC) / Instruction Pointer (IP)
    - Points to the next instruction to be executed
  - Flags
    - ALU flags: carry, overflow, zero, etc
    - Machine control flags: MMU, I/O, etc
  - Stack pointer
    - Top of the stack
    - Frame pointer
  - Return value/pointer

Data Addressing Modes

- Legend
  - 16-bit architecture
  - big-endian
  - byte-addressed memory
  - involved
  - altered

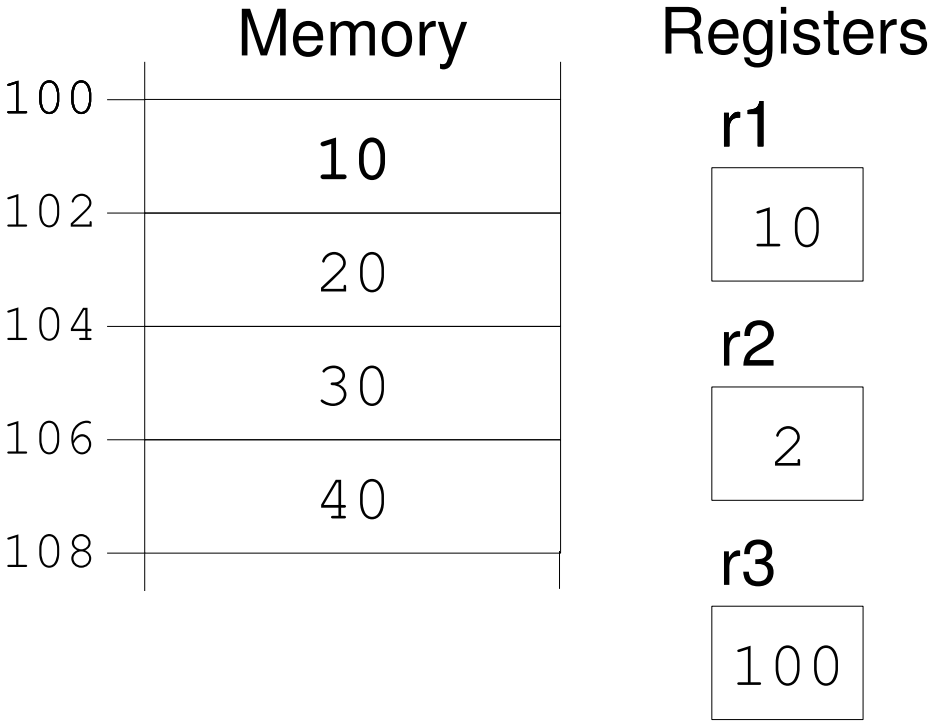


Data Addressing Modes

- Register
  - Operand stored in a CPU register
- Use
  - ALU operations
  - Most used variables
    - Automated by compilers

```
add r1, r2
```

(r1 <- r1 + r2)

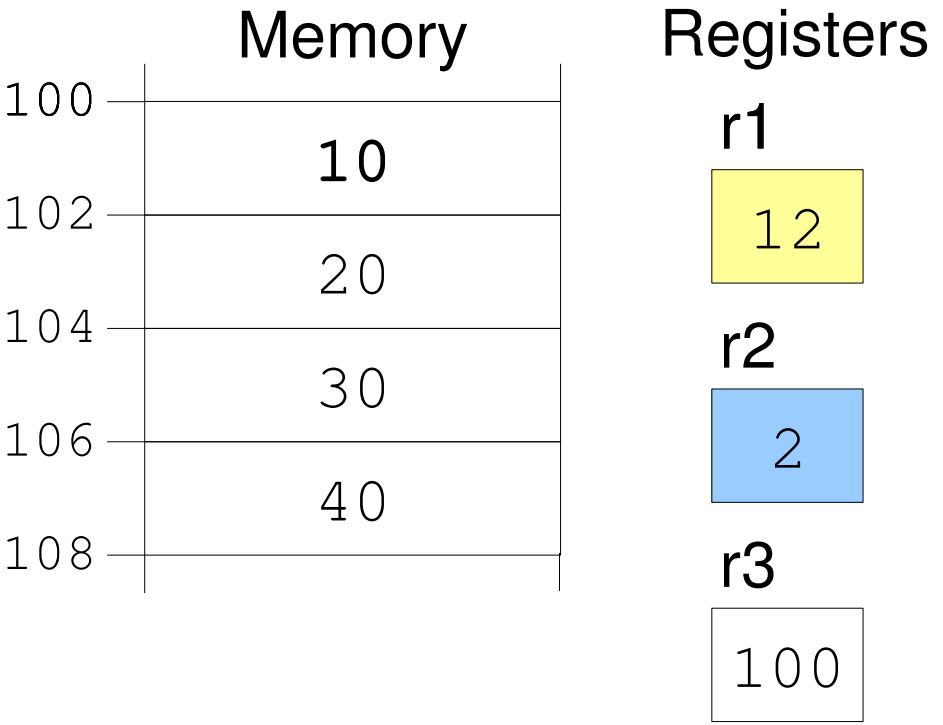


Data Addressing Modes

- Register
  - Operand stored in a CPU register
- Use
  - ALU operations
  - Most used variables
    - Automated by compilers

```
add r1, r2
```

(r1 <- r1 + r2)

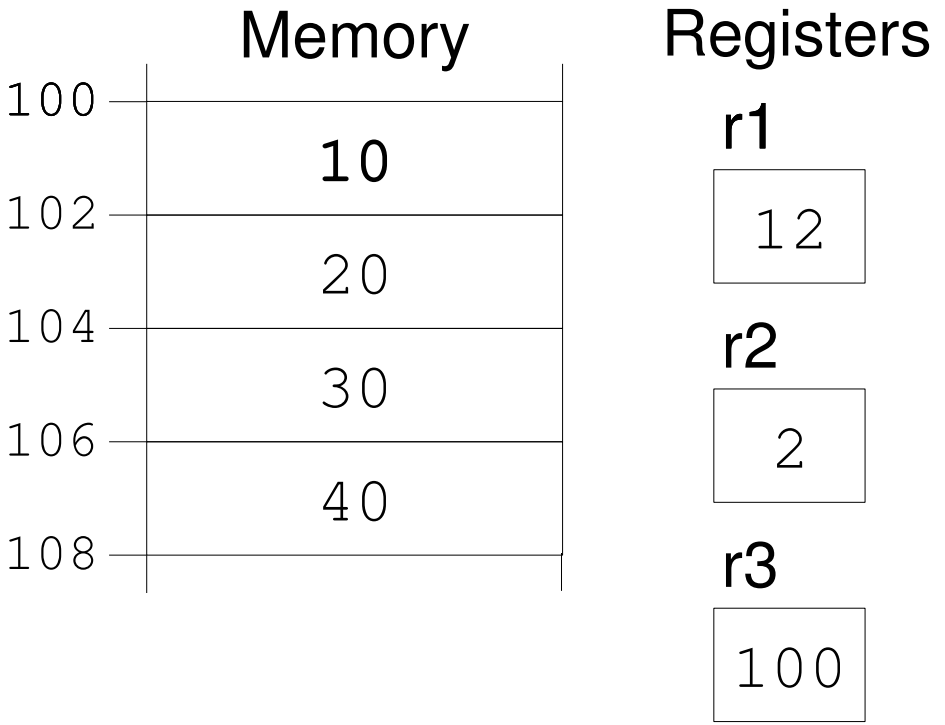


Data Addressing Modes

- Immediate
  - Operand encoded along with operation
- Use
  - Constants

```
add r1, #10
```

(r1 <- r1 + 10)

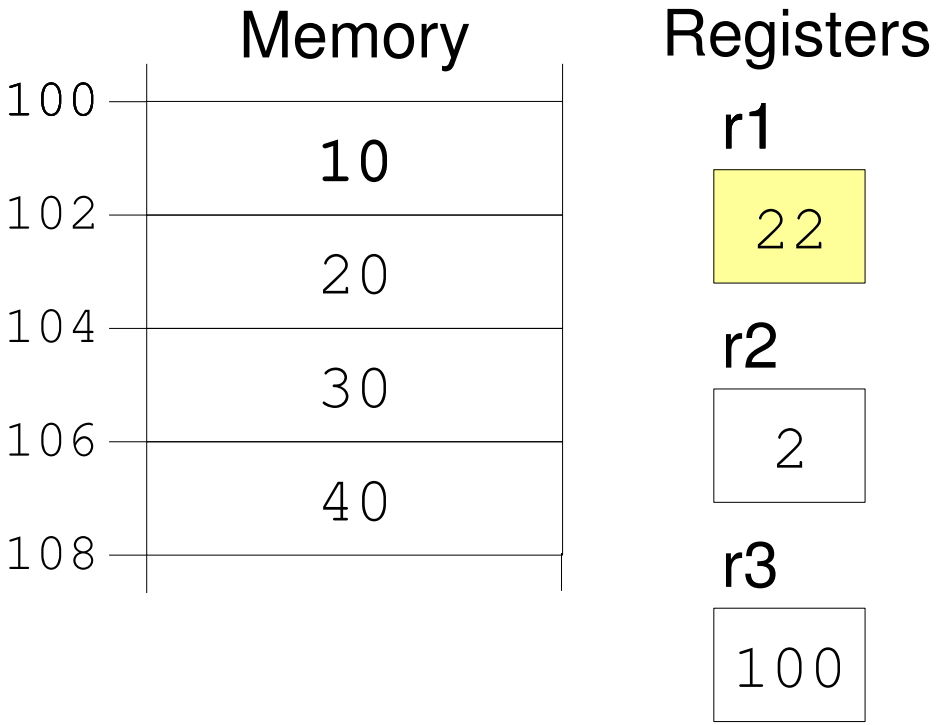


Data Addressing Modes

- Immediate
  - Operand encoded along with operation
- Use
  - Constants

```
add r1, #10
```

(r1 <- r1 + 10)

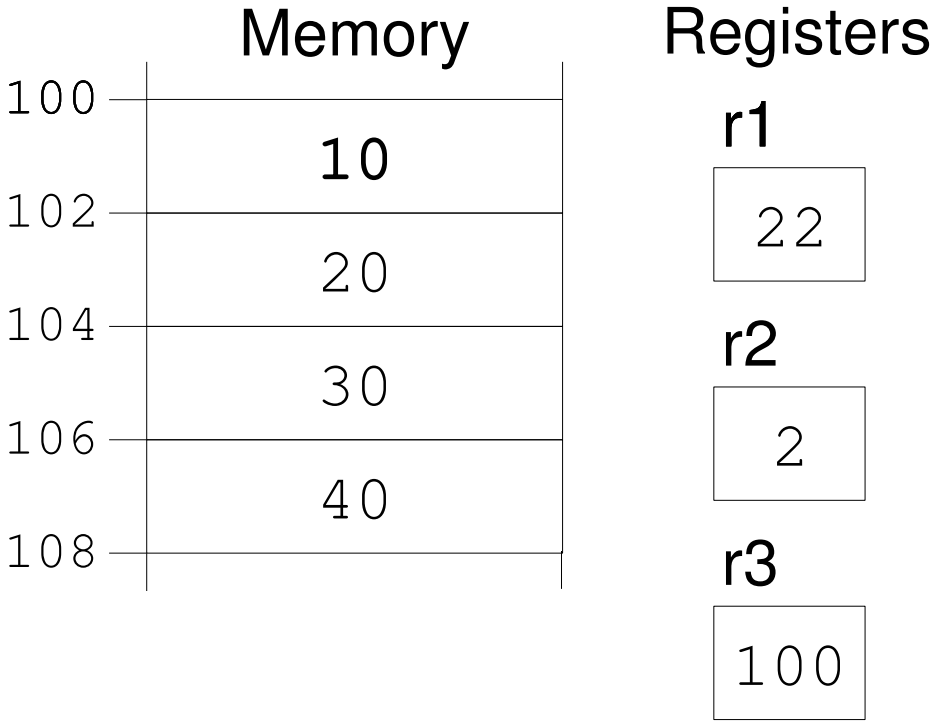


Data Addressing Modes

- Absolute
  - Operand stored in memory at `address`
- Use
  - Static data (static variables, class attributes)

`add r1, 100`

(r1 <- r1 + M[100])

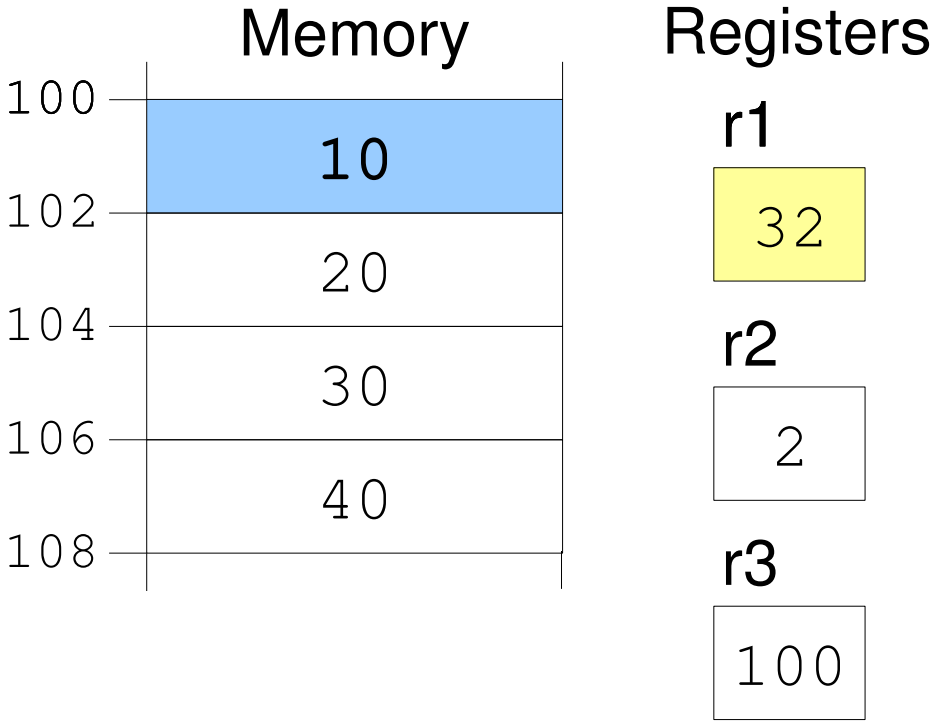


Data Addressing Modes

- Absolute
  - Operand stored in memory at `address`
- Use
  - Static data (static variables, class attributes)

`add r1, 100`

(r1 <- r1 + M[100])

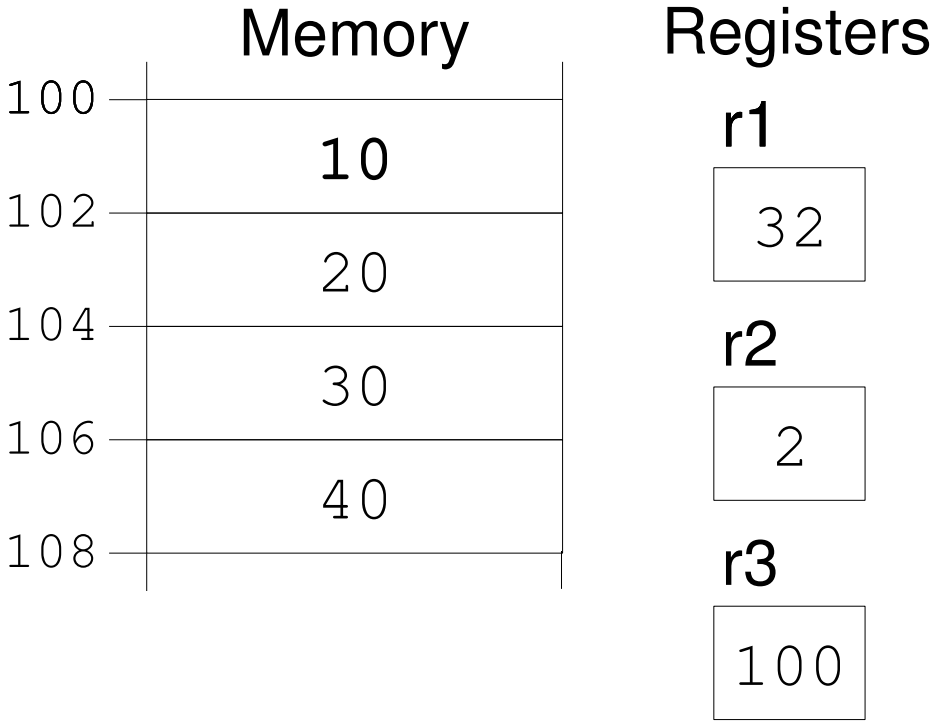


Data Addressing Modes

- Register Indirect
  - Operand stored in a memory location pointed by a `register`
- Use
  - Pointer deferring

`add r1, (r3)`

(r1 <- r1 + M[r3])

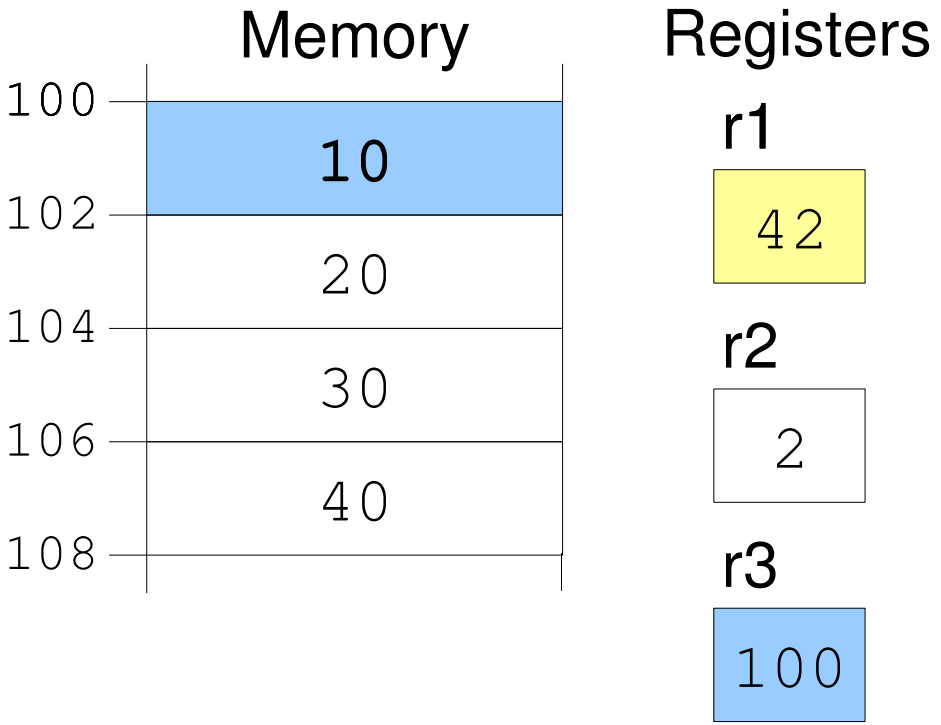


Data Addressing Modes

- Register Indirect
  - Operand stored in a memory location pointed by a `register`
- Use
  - Pointer deferring

`add r1, (r3)`

(r1 <- r1 + M[r3])

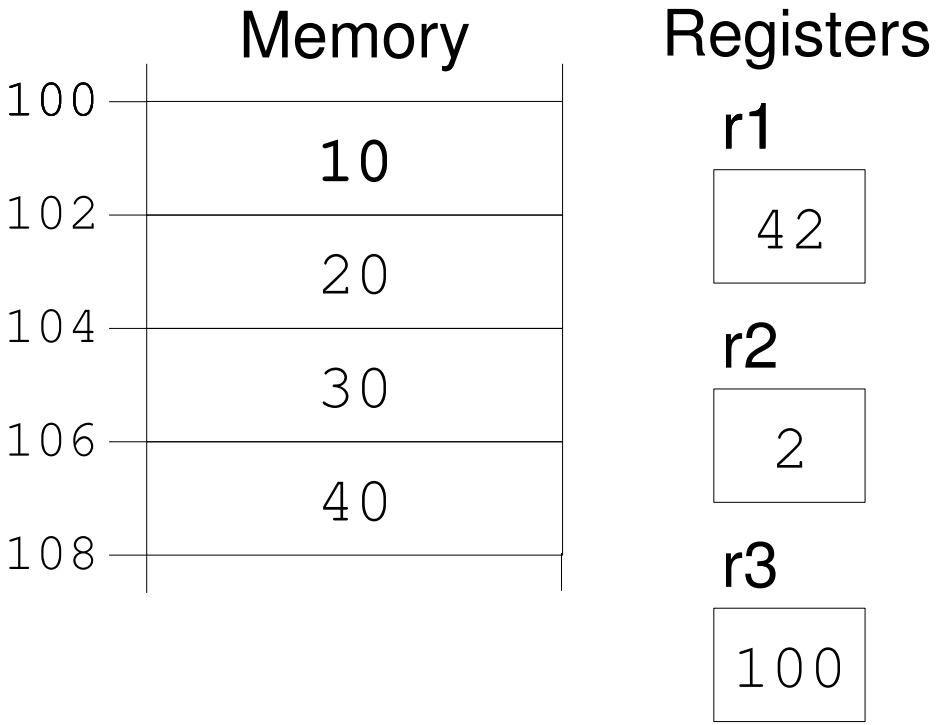


Data Addressing Modes

- Register Indirect with Displacement
  - Operand stored in a memory location pointed by a `register` plus an offset
- Use
  - Local variables (reg = frame pointer)

`add r1, 4(r3)`

(r1 <- r1 + M[r3 + 4])

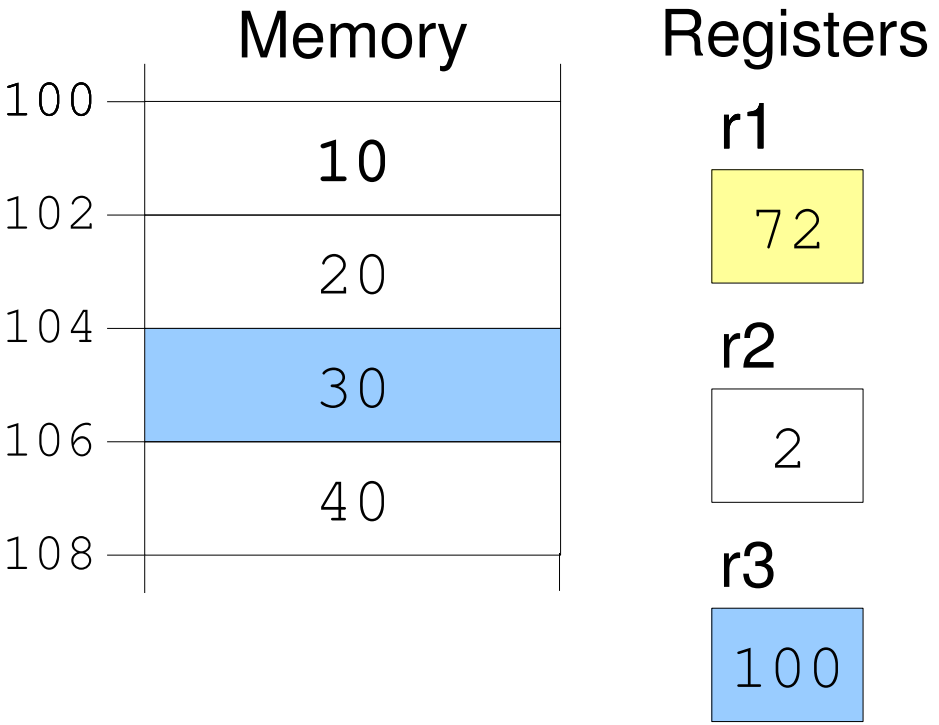


Data Addressing Modes

- Register Indirect with Displacement
  - Operand stored in a memory location pointed by a `register` plus an offset
- Use
  - Local variables (reg = frame pointer)

```
add r1, 4(r3)
```

(r1 <- r1 + M[r3 + 4])

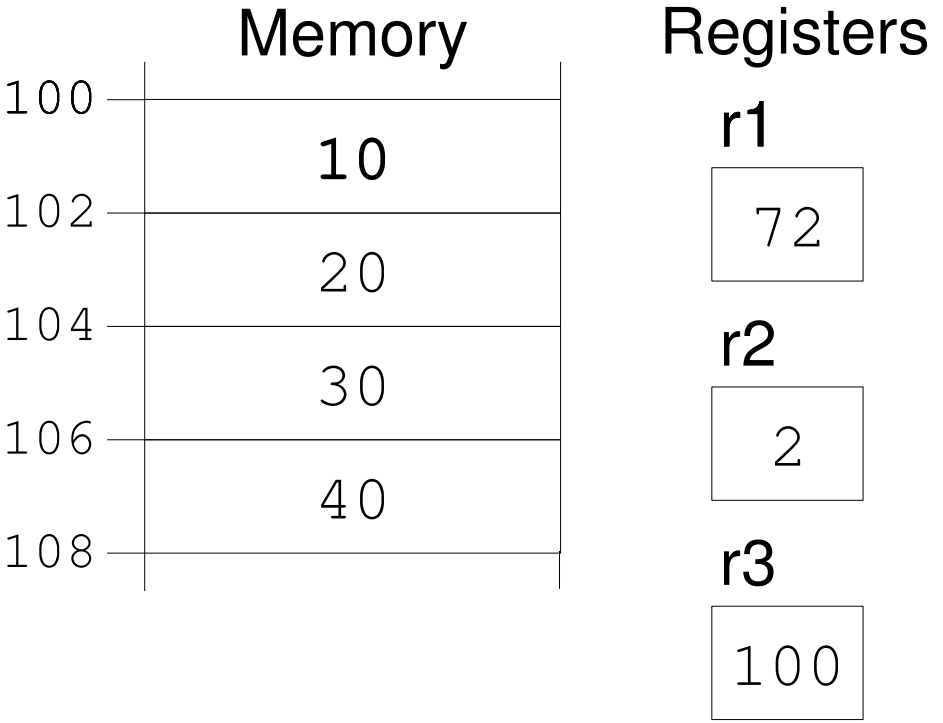


Data Addressing Modes

- Register Indirect Indexed
  - Operand stored in a memory location pointed by an indexed `register`
- Use
  - Array (reg1 = array, reg2 = index)

```
add r1, (r3, r2)
```

(r1 <- r1 + M[r3 + r2])

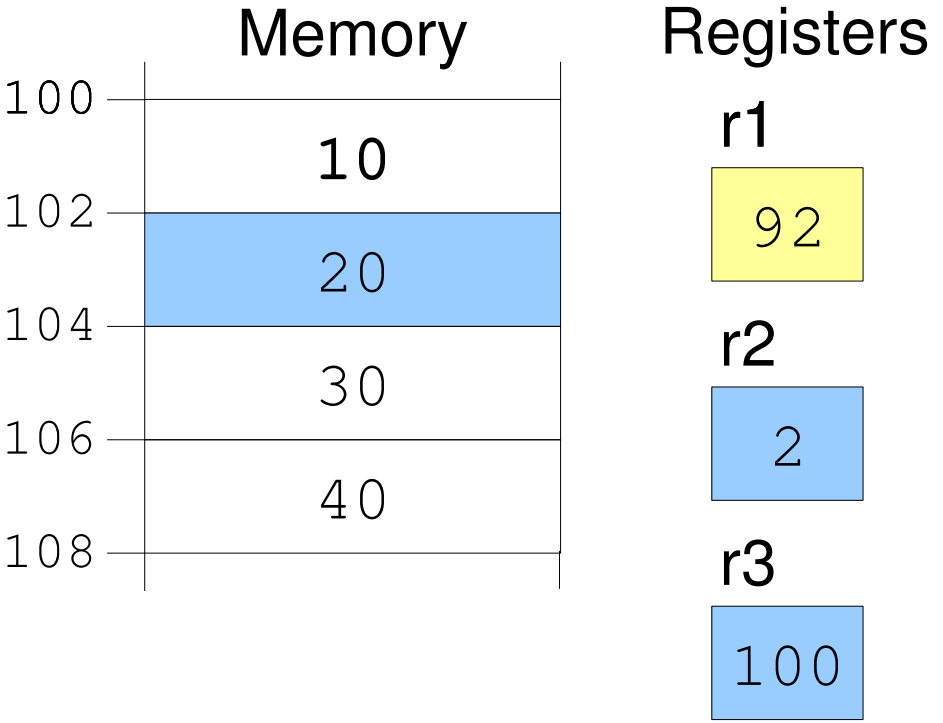


Data Addressing Modes

- Register Indirect Indexed
  - Operand stored in a memory location pointed by an indexed `register`
- Use
  - Array (reg1 = array, reg2 = index)

```
add r1, (r3, r2)
```

(r1 <- r1 + M[r3 + r2])

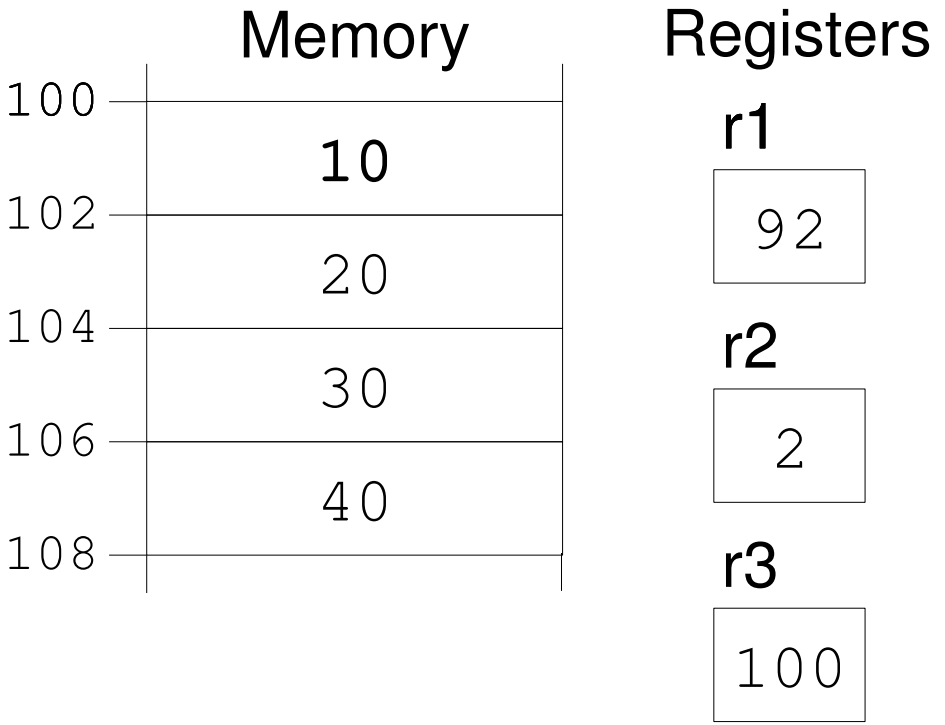


Data Addressing Modes

- Register Indirect with Post-[Inc|Dec]rement
  - Operand stored in a memory location pointed by a `register` that is post-[inc|dec]remented
- Use
  - Iteration (reg = array, inc = size of element)

`add r1, (r3)+`

(r1 <- r1 + M[r3]; r3 <- r3 + s)

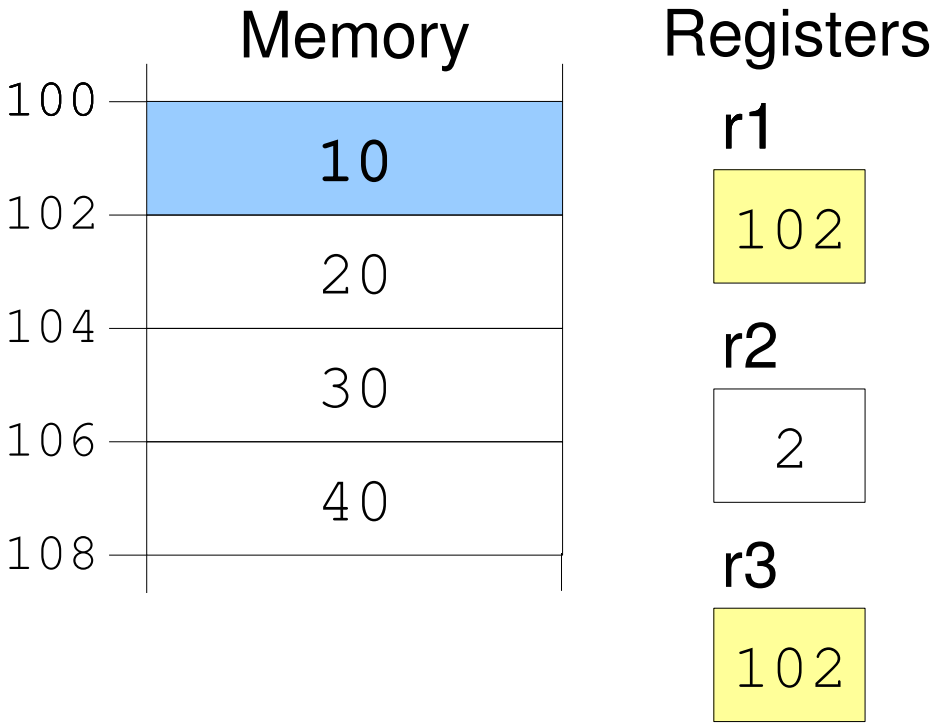


Data Addressing Modes

- Register Indirect with Post-[Inc|Dec]rement
  - Operand stored in a memory location pointed by a `register` that is post-[inc|dec]remented
- Use
  - Iteration (reg = array, inc = size of element)

`add r1, (r3)+`

(r1 <- r1 + M[r3]; r3 <- r3 + s)

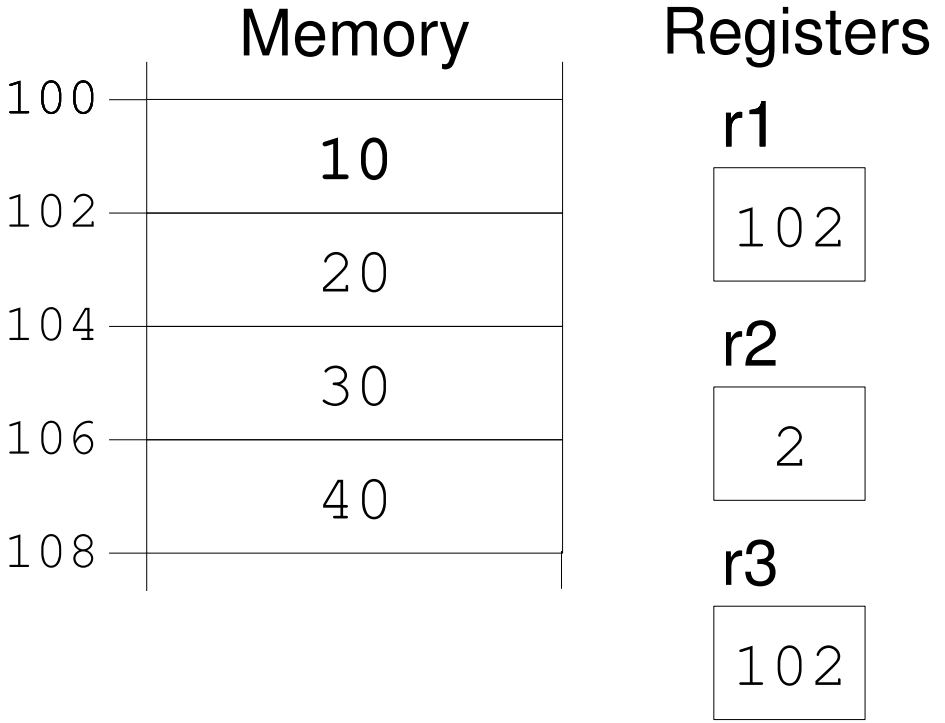


Data Addressing Modes

- Register Indirect with Pre-[Inc|Dec]rement
  - Operand stored in a memory location pointed by a `register` that is pre-[inc|dec]remented
- Use
  - Iteration
  - Stack (reg = stack ptr)

`add r1, -(r3)`

(r3 <- r3 - s; r1 <- r1 + M[r3])

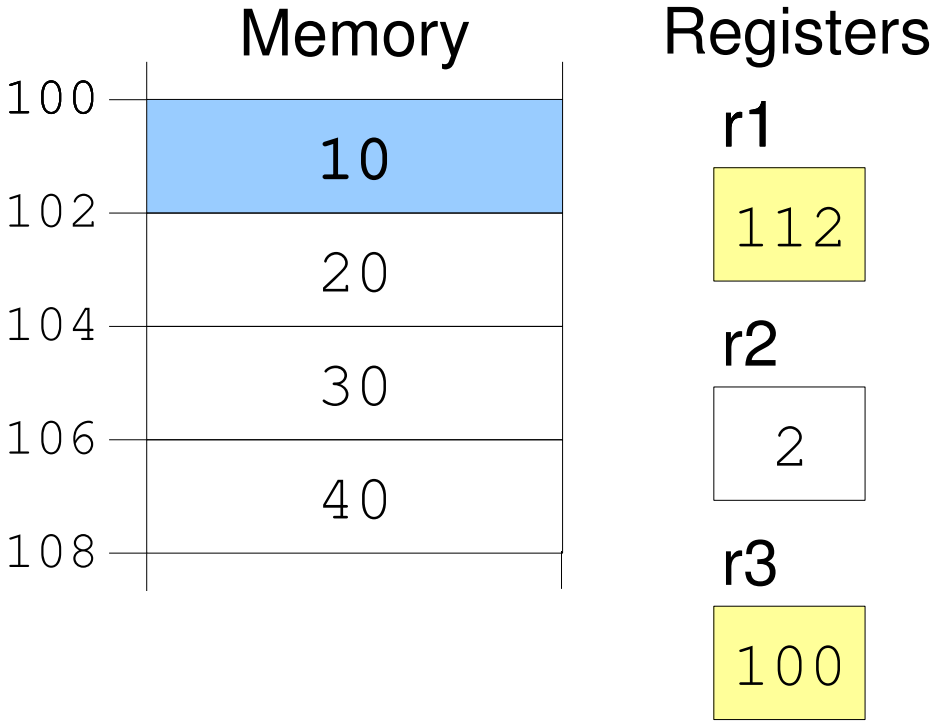


Data Addressing Modes

- Register Indirect with Pre-[Inc|Dec]rement
  - Operand stored in a memory location pointed by a `register` that is pre-[inc|dec]remented
- Use
  - Iteration
  - Stack (reg = stack ptr)

`add r1, -(r3)`

(r3 <- r3 - s; r1 <- r1 + M[r3])

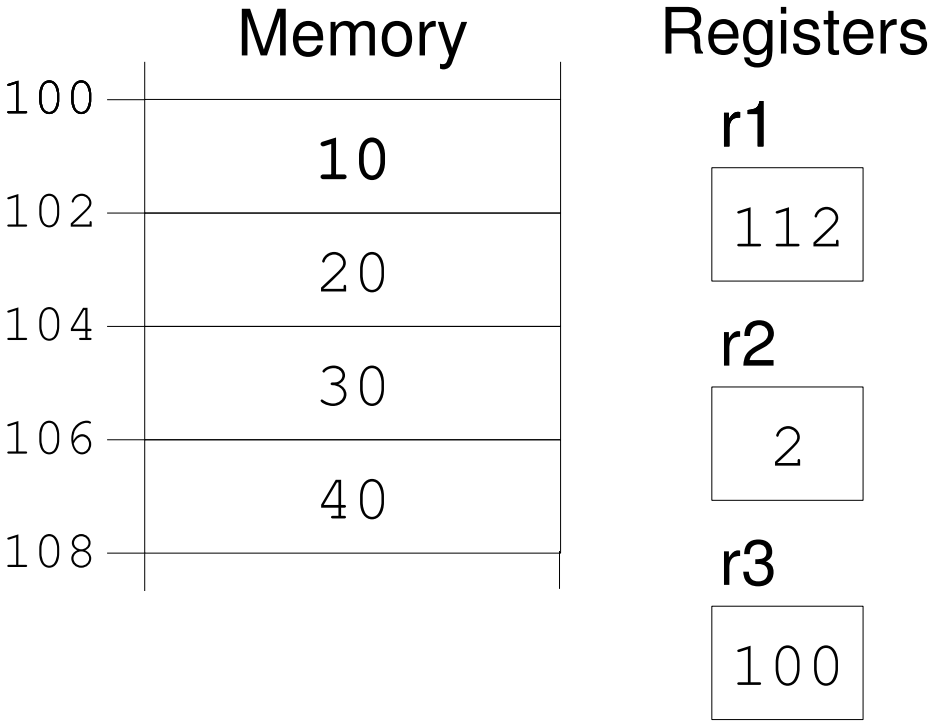


Data Addressing Modes

- Register Indirect Indexed and Scaled
  - Operand stored in a memory location pointed by an indexed and scaled `register`
- Use
  - Array (reg1 = array, reg2 = index, scale = size of element)

```
add r1, (r3, r2, 2)
```

(r1 <- r1 + M[r3 + r2 * 2)

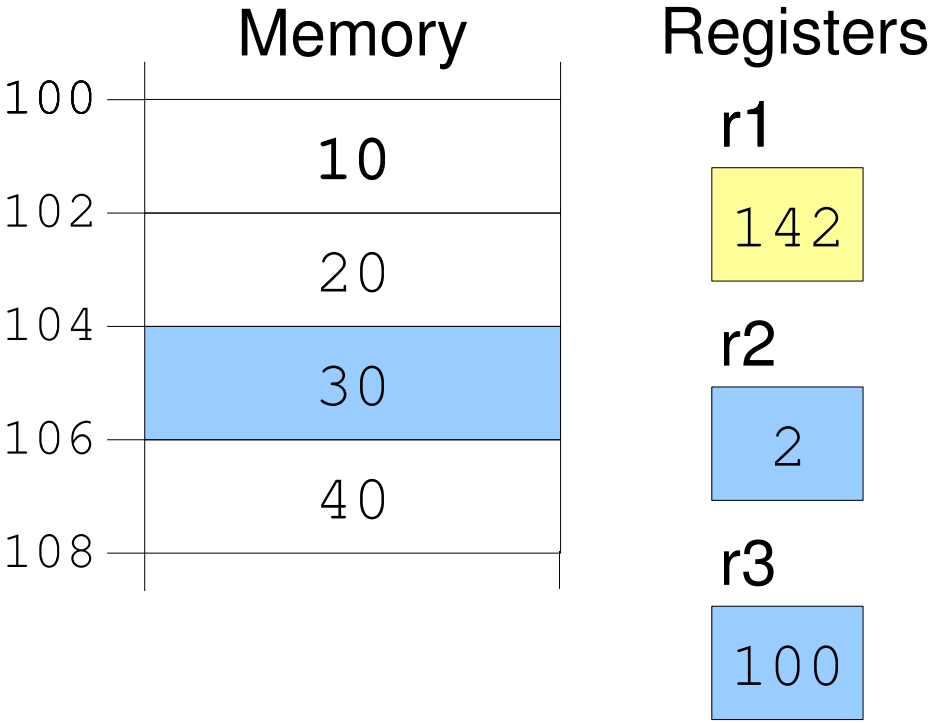


Data Addressing Modes

- Register Indirect Indexed and Scaled
  - Operand stored in a memory location pointed by an indexed and scaled `register`
- Use
  - Array (reg1 = array, reg2 = index, scale = size of element)

```
add r1, (r3, r2, 2)
```

(r1 <- r1 + M[r3 + r2 * 2)

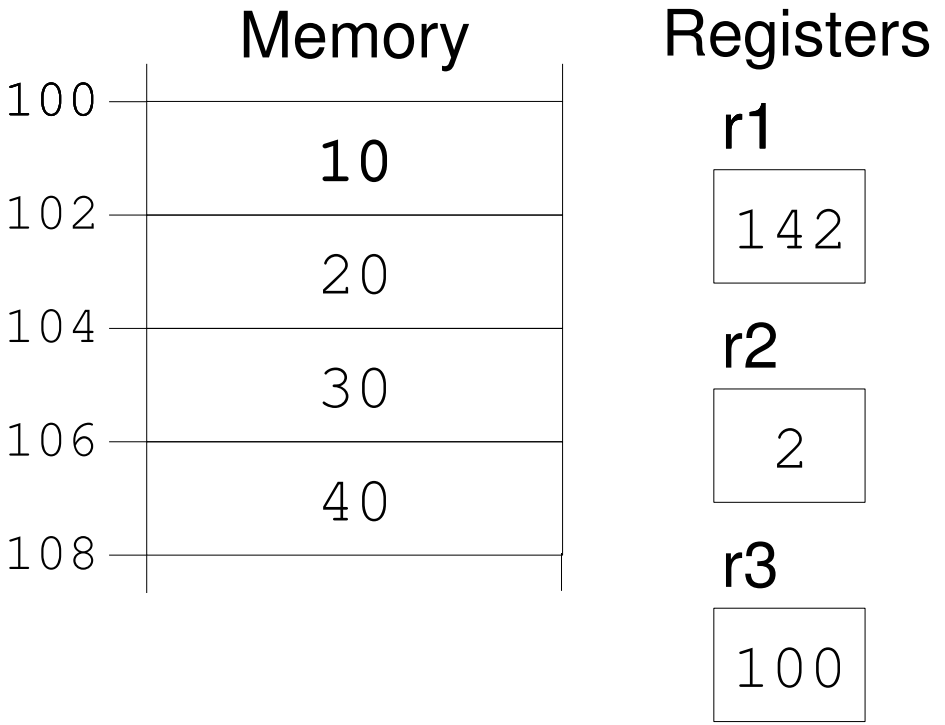


Data Addressing Modes

- Register Indirect with Displacement, Indexed and Scaled
- Use
  - Array (dis = offeset, reg1 = array, reg2 = index, scale = size of element)

```
add r1, 2(r3, r2, 2)
```

(r1 <- r1 + M[r3 + 2 + r2 * 2])



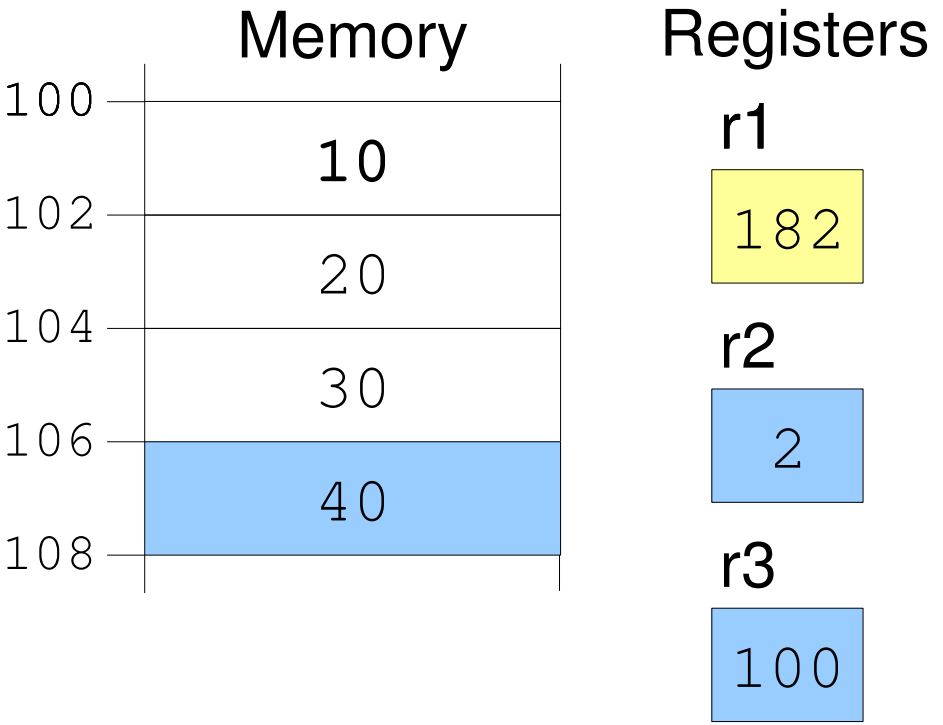
Data Addressing Modes

- Register Indirect with Displacement, Indexed and Scaled

```
add r1, 2(r3, r2, 2)
```

(r1 <- r1 + M[r3 + 2 + r2 * 2])

- Use
  - Array (dis = offeset, reg1 = array, reg2 = index, scale = size of element)

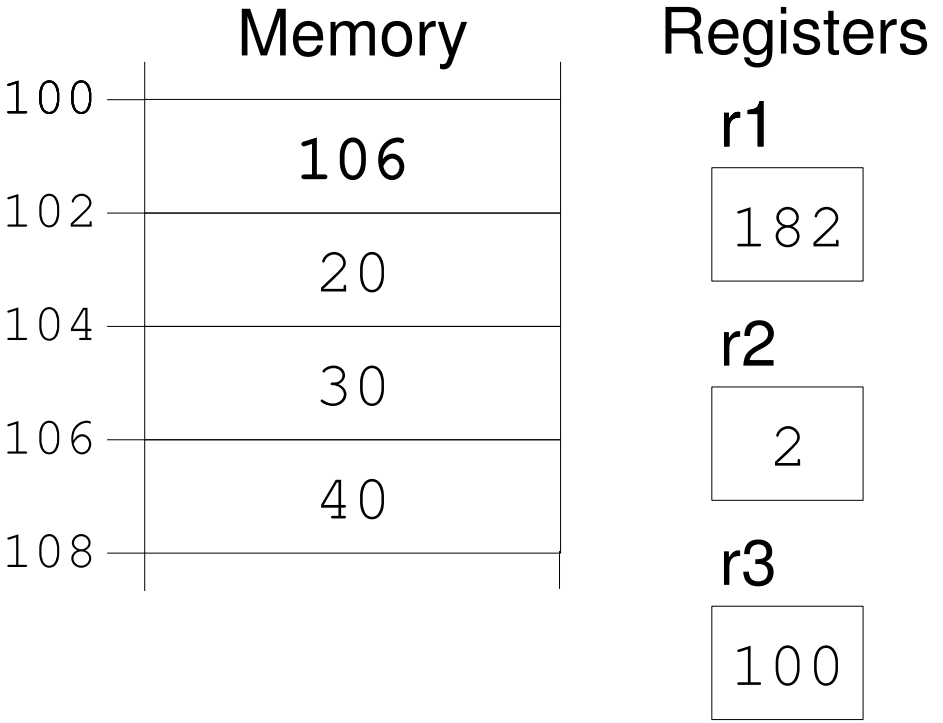


Data Addressing Modes

- Memory Indirect
  - Operand stored in a memory location pointed by another `memory` localion
- Use
  - Pointer to pointer deferring
  - Call tables

```
add r1, @(r3)
```

(r1 <- r1 + M[M[r3]])

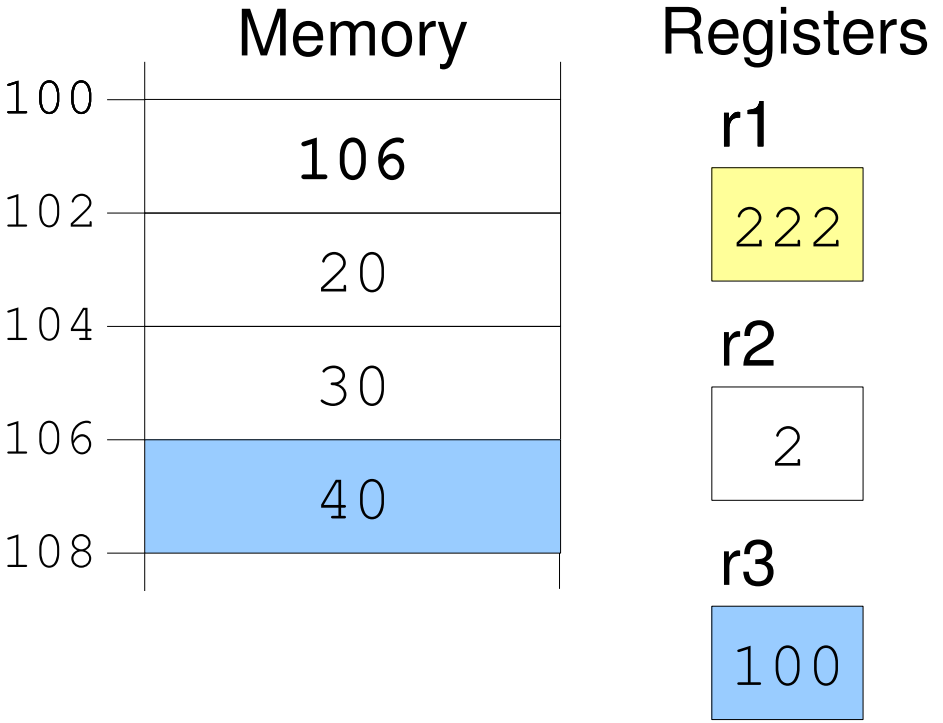


Data Addressing Modes

- Memory Indirect
  - Operand stored in a memory location pointed by another `memory` localion
- Use
  - Pointer to pointer deferring
  - Call tables

`add r1, @(r3)`

(r1 <- r1 + M[M[r3]])



Data Addressing Modes

Mode	Example	Meaning
Register	`add r1, r2`	r1 <- r1 + r2
Immediate	`add r1, #10`	r1 <- r1 + 10
Absolute	`add r1, 100`	r1 <- r1 + M[100]
Reg. Ind. Disp.	`add r1, (r3)`	r1 <- r1 + M[r3]
Reg. Ind. Disp.	`add r1, 4(r3)`	r1 <- r1 + M[r3 + 4]
Reg. Ind. Ind.	`add r1, (r3, r2)`	r1 <- r1 + M[r3 + r2]
Reg. Ind. Post-	`add r1, (r3)+`	r1 <- r1 +M[r3]; r3 <- r3 + s
Reg. Ind. Pre-	`add r1, -(r3)`	r3 <- r3 - s; r1 <- r1 +M[r3]
Reg. Ind. Scal.	`add r1, (r3, r2, 2)`	r1 <- r1 + M[r3 + r2 * 2]
Reg. Ind. Disp. Scal.	`add r1, 2(r3, r2, 2)`	r1 <- r1 + M[r3 +2 + r2 * 2]
Memory Ind.	`add r1, @(r3)`	r1 <- r1 + M[M[r3]]

Arithmetic and Logical Operations

Hugo Marcondes
hugom@lisha.ufsc.br
http://www.lisha.ufsc.br/~hugom

Sep 2006

Introduction

- Logical Operations
  - Boolean Algebra
    - AND, OR, NOR, XOR
  - Shift Operations
    - Shifts Right/Left
    - Rotates Right/Left
- Arithmetic Operations
  - Sum and Subtraction
  - Multiplication and Division

Logical Operations

- Essential for low level programming
  - Device Drivers
  - Data Transmission (Serial/Parallel)
  - Data Codification
  - Cryptography Algorithms
- Bits are manipulated with bit masks

Logical Operations

- AND Function

  and rd, rs, rt

- Use
  - Verify if a bit is set
  - Unset a bit

- Alternate form

  andi rd, rs, imm

RD	RS	RT
0	0	0
0	0	1
0	1	0
1	1	1

Logical Operations

- OR Function

  or rd, rs, rt

- Use
  - Set a bit
  - Load constants

- Alternate form

  ori rd, rs, imm

RD	RS	RT
0	0	0
1	0	1
1	1	0
1	1	1

Logical Operations

- Negate-OR Function

  nor rd, rs, rt

- Use
  - Detect both zero bits
  - Invert bits (all)

RD	RS	RT
1	0	0
0	0	1
0	1	0
0	1	1

Logical Operations

- Exclusive-OR Function

  xor rd, rs, rt

- Use
  - Detect positions with different bits
  - Invert bits (mask)

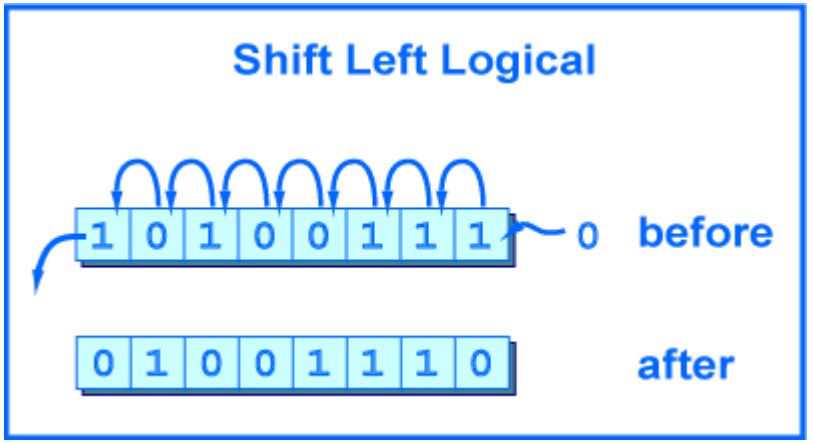
RD	RS	RT
0	0	0
1	0	1
1	1	0
0	1	1

Logical Operations

- Left Shift Operations
  sll rd, rt, shamt

- Use
  - Bit level Operations
  - Multiplication and division
  - Serialization / Parallelization

- Variable shift
  sllv rd, rt, rs

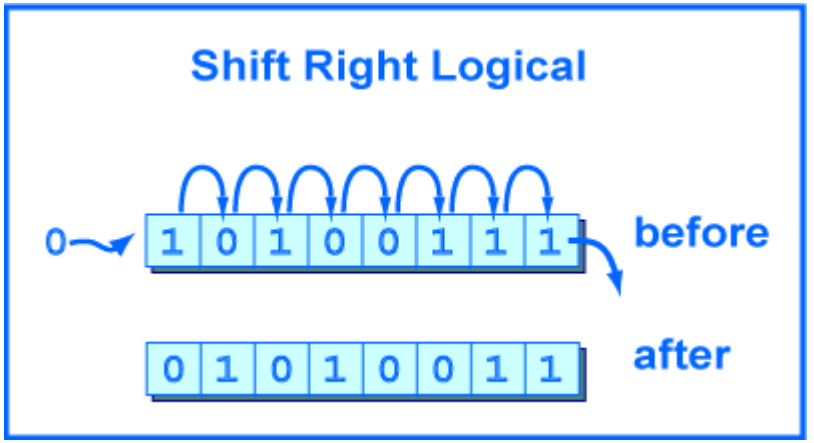


[Kjell, 2004]

Logical Operations

- Right Shift Operations
  srl rd, rt, shamt

- Use
  - Bit level Operations
  - Multiplication and division
  - Serialization / Parallelization
- Variable shift
  srlv rd, rt, rs
- Arithmetic shift
  sra rd, rt, shamt / srav rd, rt, rs



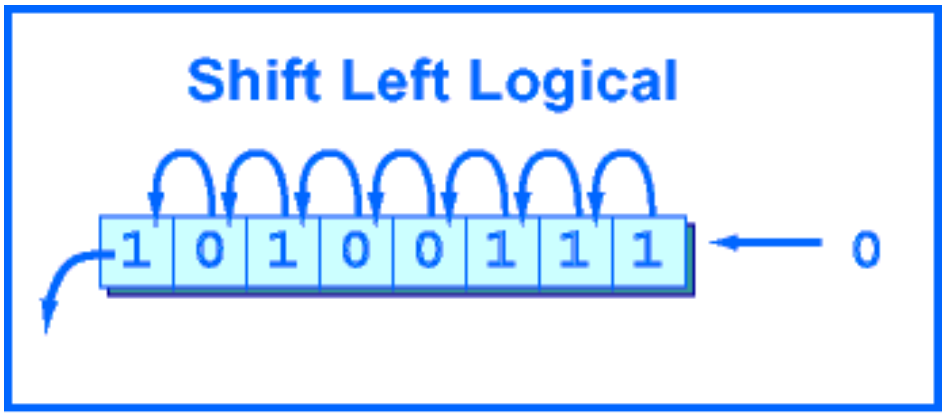
[Kjell, 2004]

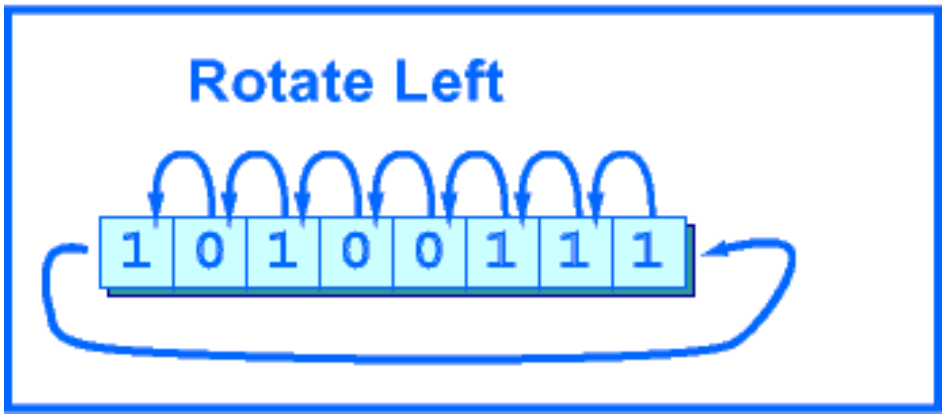
Logical Operations

- Right/Left Rotation
  - Bits aren't discarded
- Right Rotation

  High bit moves to low
- Left Rotation

  Low bit moves high





[Kjell, 2004]

Arithmetic Operations

- Sum
  - With or without overflow signaling
  - Can have a immediate operand
- Subtraction
  - Sum of a negative number

- Overflow signalization

Operation	A	B	RES
A + B	>= 0	>= 0	< 0
A + B	< 0	< 0	>= 0
A – B	>= 0	< 0	< 0
A – B	< 0	>= 0	>= 0

Arithmetic Operations

- Comparison Instructions
  - Arithmetical comparison between number
  - Less than, Greater Than, Equal, Not Equal
- The result of comparison can be in a general purpose register, or specific ones
  - Intel: Flags register
  - MIPS

    Set on Less Than

    slt rd, rs, rt / sltu rd, rs, rt

    slti rd, rs, imm / sltiu rd, rs, imm

    - Pseudo instructions for other modes

Arithmetic Operation

- Multiplication
- Product of two N-bits number is 2N bits

```
      10
 x    15
-------
      50
     10  +
-------
     150
```

```
      1010
 x    1111
--------
      1010
     1010
    1010
   1010   +
--------
10010110
```

Arithmetic Operation

- Result are stored on a double word register
- MIPS has HI and LO register
  - Accessed mflo rd and mfhi rd instructions

  mult rs, rt and multu rs, rt

  $HI_{32}LO_{32}$ <- rs * rt
- Unsigned Multiplication
  - Overflow don't occur
- Signed Multiplication
  - Equal signs: positive result
  - Can produce overflow
  - MIPS need overflow detection on software

Arithmetic Operation

- Division

  div rs, rt and divu rs, rt

  LO <- Quotient (rs/rt) , HI <- Remainder

  - Mathematical Properties
    - Don't generate overflow
    - Raises a Zero Division exception
    - Result is always unsigned

System Programming
Flow Control

Hugo Marcondes
hugom@lisha.ufsc.br
http://www.lisha.ufsc.br/~hugom

Sep 2006

Introduction

"The power of computers is their ability to repeat actions and their ability to alter their operation depending on data"

- Bradley Kjell

- Instruction Addressing Modes
  - Branchs
    - Conditional
    - Unconditional
  - Long Branch
    - Jump
    - Call

Instruction Addressing Modes

- Branch to an absolute address

  Concatenates imm with high bits of PC to complete address

- Use

  Calling near functions

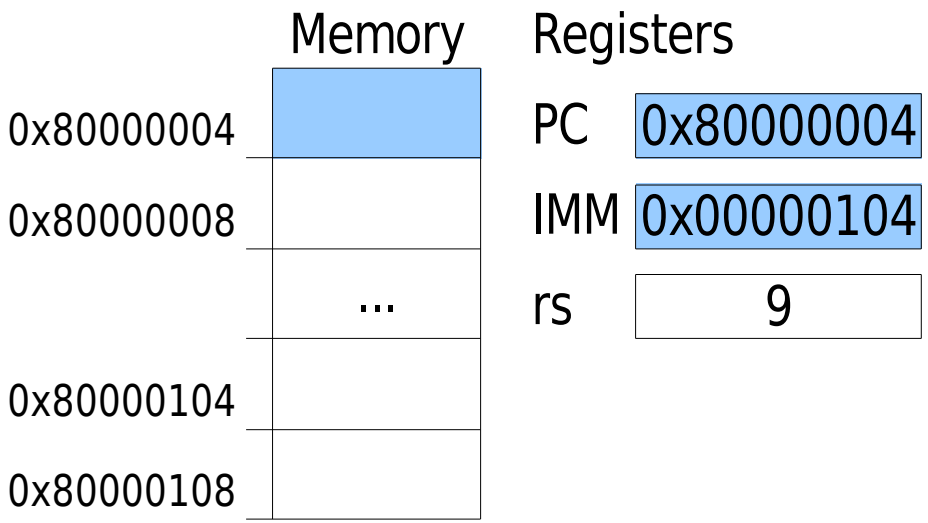
Absolute

jump $imm_{[n]}$

$PC <- PC_{[w,n]} : imm_{[n]}$

w -> word High Bit

n -> imm High Bit



Instruction Addressing Modes

- Branch to an absolute address

  Concatenates imm with high bits of PC to complete address

- Use

  Calling near functions

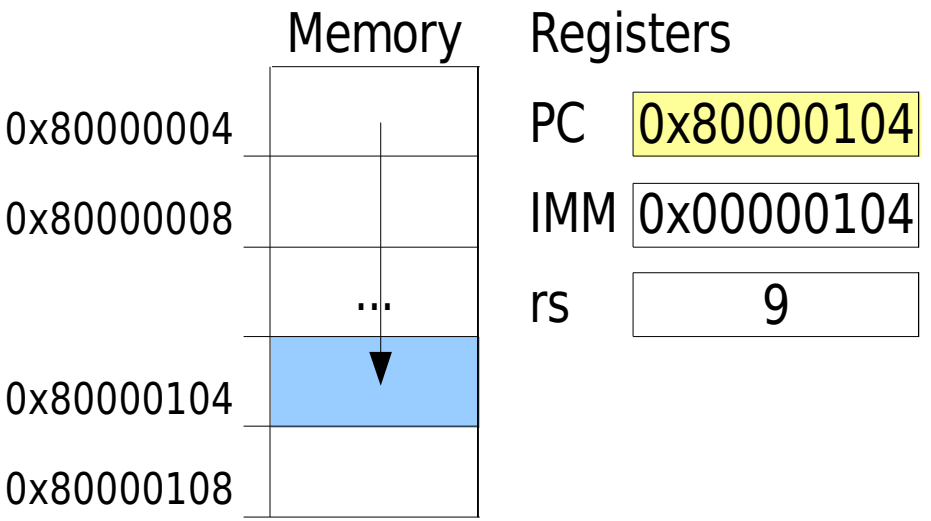
Absolute

jump $imm_{[n]}$

$PC <- PC_{[w,n]} : imm_{[n]}$

w -> word High Bit

n -> imm High Bit



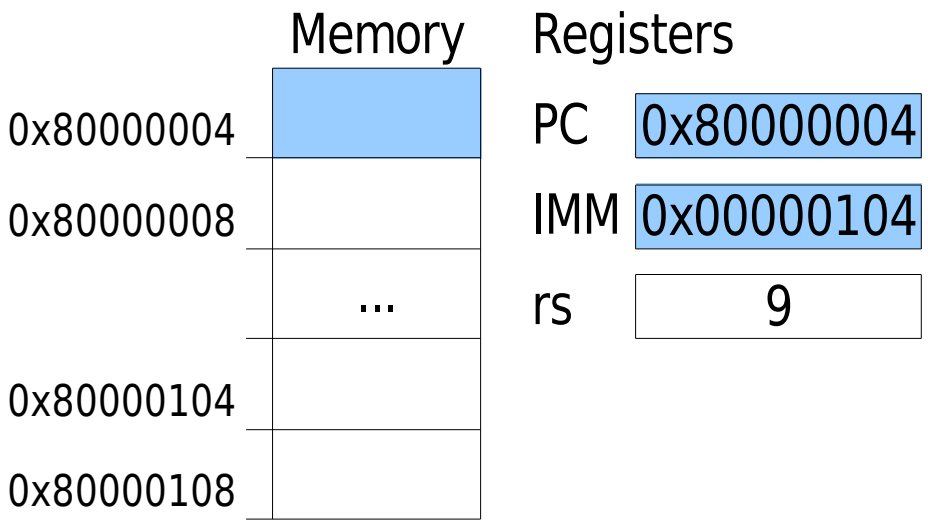
Instruction Addressing Modes

- Branch to a relative address
  - Sum imm with the PC
  - Usually imm is signed
  - Code can be easily relocated
- Use
  Local Branch

Relative Address

branch imm

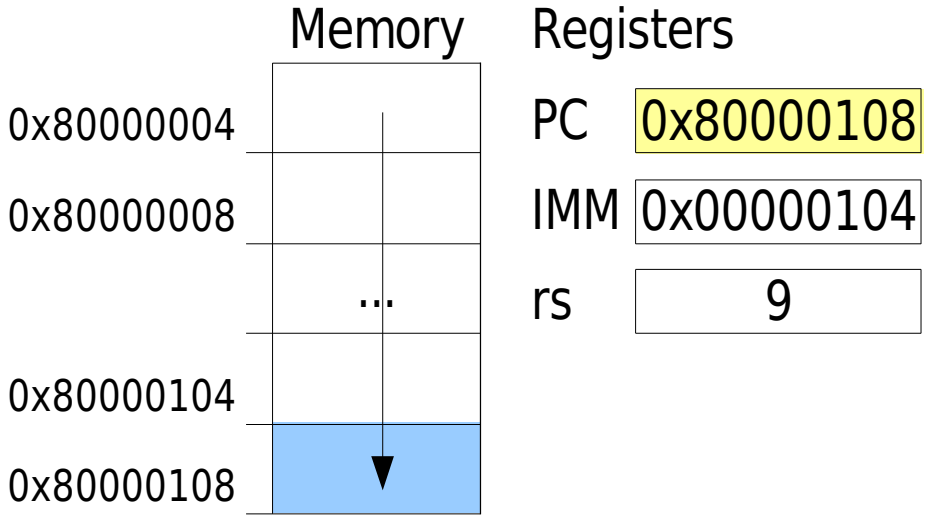
PC <- PC + imm



Instruction Addressing Modes

- Branch to a relative address
  - Sum imm with the PC
  - Usually imm is signed
  - Code can be easily relocated
- Use
  Local Branch

Relative Address
branch imm
PC <- PC + imm



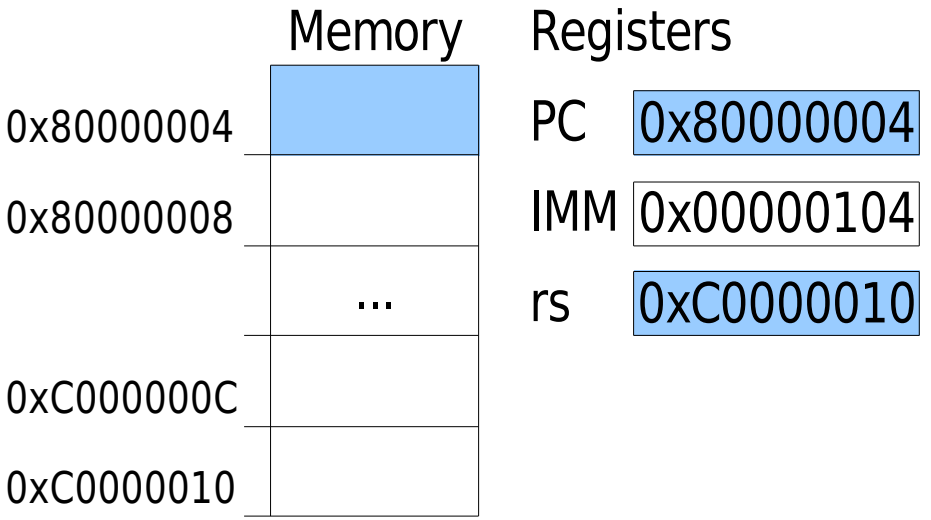
Instruction Addressing Modes

- Branch to an absolute address
  - Contents of rs are transferred to PC
- Use
  Long Branch

Register Indirect

jump rs

PC <- rs



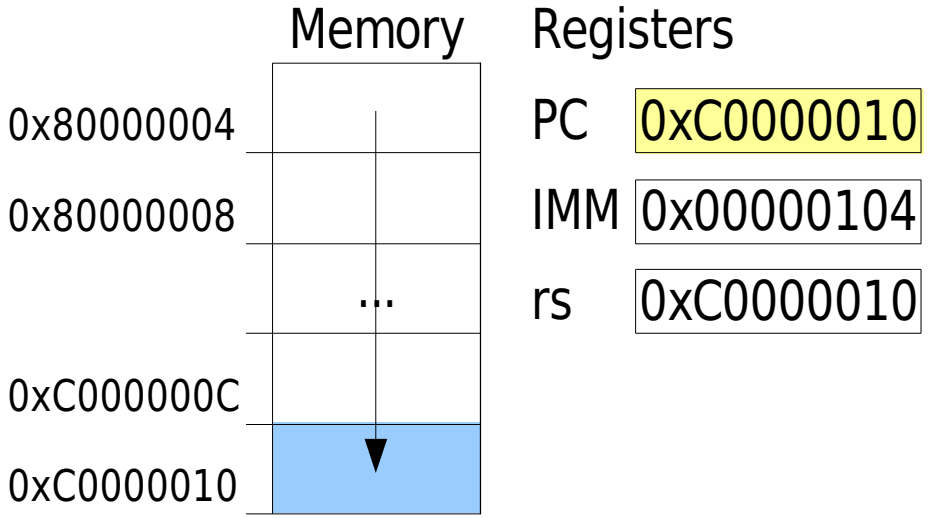
Instruction Addressing Modes

- Branch to an absolute address
  - Contents of rs are transferred to PC
- Use
  Long Branch

Register Indirect

jump rs

PC <- rs



Branch Instructions

- Instructions used with address modes to change the execution flow of a program
- Conditional Branchs
  - usually associated with Relative Addressing Mode
  - Have a condition associated with the branch
  - Comparison instructions/registers to express the condition
- Unconditional Branch
  - Usually have all addressing mode forms
  - Always change the execution flow
  - Used in local context or for function call

Conditional Branch

- Condition expressed in the instruction

  beq rs, rt, LABEL (Branch Equal)

  PC <- PC + LABEL, if rs == rt

  bne rs, rt, LABEL (Branch Not Equal)

  PC <- PC + LABEL, if rs != rt

- Condition resolved before the branch instruction

  slt rd, rs, rt (Set On Less Than)

  rd <- 1, if rs < rt

  rd <- 0, otherwise

  Used in conjunction with beq / bne

Unconditional Branch

- Are not associated with a condition

b LABEL (Branch to LABEL – Relative Address)

PC <- PC + LABEL

j ADDR (Jump to ADDR – Absolute Address)

PC <- PC : ADDR

jr rs (Jump Register – Register Indirect)

PC <- rs

Calls

- Branch the execution flow, but saves the return address
- The return address could be saved in stack or in a specific purpose register

jal LABEL (Jump and Link – Relative Address)

RA <- PC + 4 ; PC <- PC : LABEL

jalr rs (Jump and Link Register – Register Indirect)

RA <- PC + 4 ; PC <- RS

Programming Structures

Exercise

Write the assembler code that maps the following programming structure

```
if ( a == 0 ) {
  // Block 1
} else {
  // Block 2
}

while (a > 0) {
  //Block 3
  a = a - 1;
}
```