Advanced Computer Architecture

Instruction Sets: MIPS

Slides prepared from the original slides of Hennessy and Patterson

Computer Science

This Lecture

- RISC vs. CISC
- Starting MIPS ISA

Instruction Sets

- Instruction: The word of a computer's language.
- Instruction set: The set or vocabulary of all the instructions.
- Instruction sets are the attributes visible to the programmer.
- They can broadly be classified as
  - Reduced Instruction Set Computing (RISC)
  - Complex Instruction Set Computing (CISC)

RISC vs. CISC

- Reduced Instruction Set Computing
- Software centric approach
- Instructions are simple (fixed size)
- A reduced number of instructions
- Instructions on average take very few cycles (single cycle)

- Complex Instruction Set Computing
- Hardware centric approach
- Instructions are complex (different sizes)
- A large number of instructions
- Instructions on average take a large number of cycles

MIPS

- Acronym for Million Instructions per Second
- Developed at Stanford by John L. Hennessy and his team
- RISC
- Used in embedded devices
- Used very frequently for educational purposes

MIPS Arithmetic Instructions

- Each MIPS arithmetic instruction
  - performs only one operation and
  - must always have three variables (variables?).

MIPS add

- 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
```

MIPS add Example from C with Multiple Operands

- C code a = b + c + d + e;
- translates into the following assembly code:

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

- Instructions are simple: fixed number of operands (unlike C)
- A single line of C code is converted into multiple lines of
- assembly code

MIPS Subtract

C code f = (g + h) – (i + j);
translates into the following assembly code:

```
add  f, g, h
sub   f, f, i
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 register and operate on the registers
- To simplify the instructions, MIPS 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… The number of registers is limited

Registers in MIPS

- The MIPS ISA has 32 registers (x86 has 8 registers)
- Each register is 32-bit 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)… (Complete set of registers later)

MIPS Add Using Registers

- C code: a = b + c ;

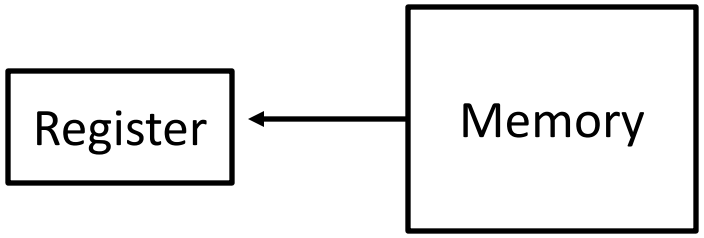
- Assembly code: (human-friendly machine instructions)
  add $s0, $s1, $s2 # assuming $s0, $s1, $s2
  # corresponds to a,b,c respectively

- Machine code: (hardware-friendly machine instructions)
  00000010001100100000000100000

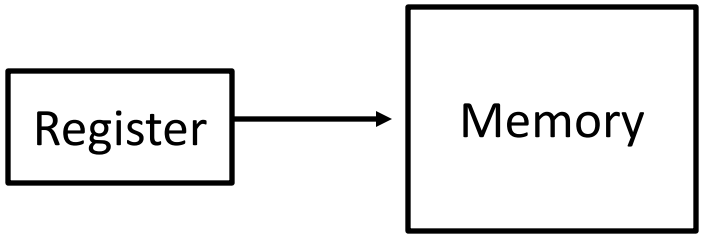
Memory Operands

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

Load word
lw $t0, memory-address



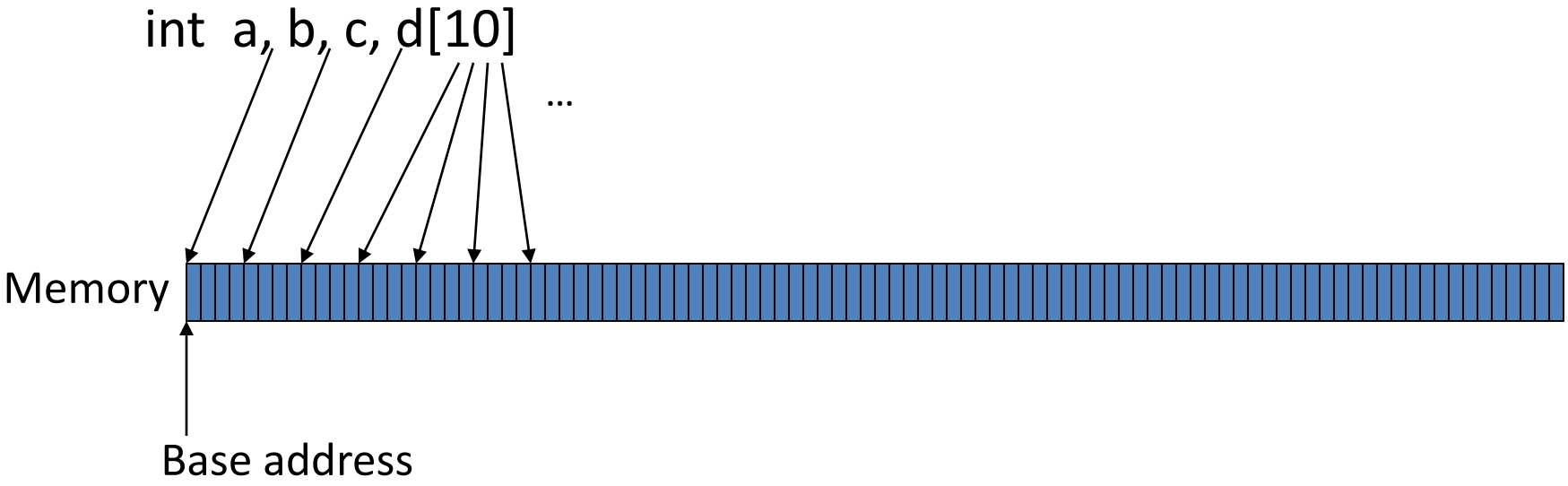
Store word
sw $t0, memory-address



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



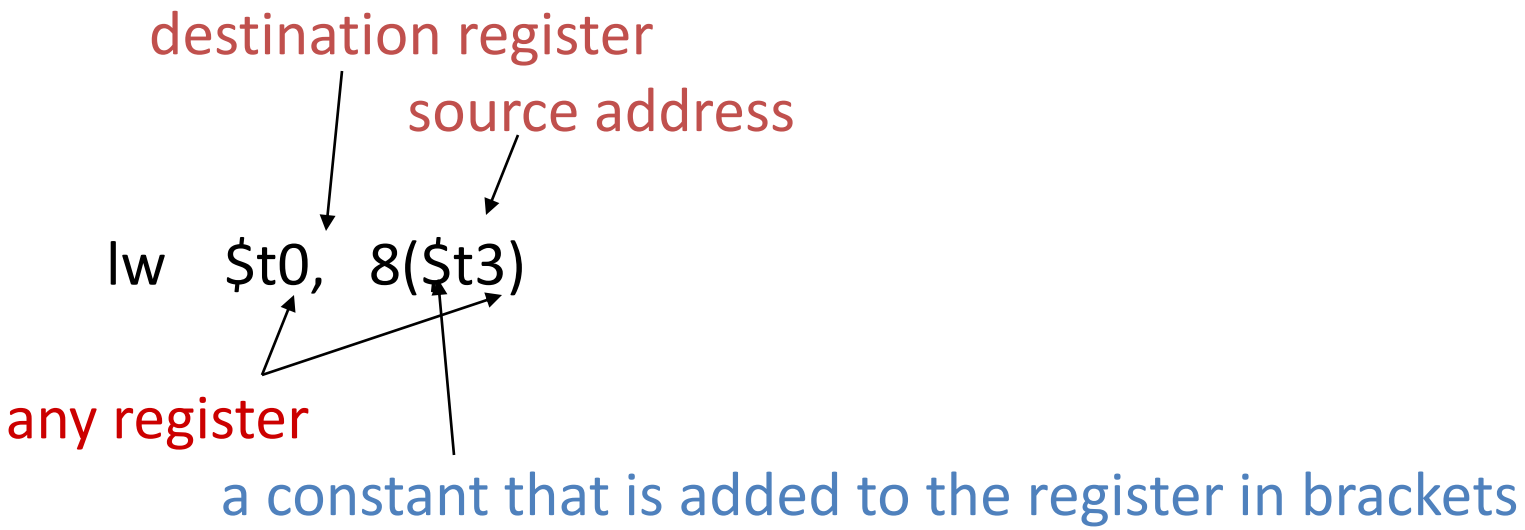
Immediate Operands

- An instruction may require a constant as input
- An immediate instruction uses a constant number as one of the inputs (instead of a register operand)

```
addi   $s0, $zero, 1000   # the program has base address
                          #  1000 and this is saved in $s0
                          # $zero is a register that always
                          # equals zero
addi   $s1, $s0, 0        # this is the address of variable a
addi   $s2, $s0, 4        # this is the address of variable b
addi   $s3, $s0, 8        # this is the address of variable c
addi   $s4, $s0, 12       # this is the address of variable d[0]
```

Memory Instruction Format

- The format of a load instruction:



Example

Convert to assembly:

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

Example

Convert to assembly:

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

Assembly: # addi instructions as before

```
lw    $t0, 8($s4)    #  d[2] is brought into $t0
lw    $t1, 0($s1)    #   a  is brought into $t1
add   $t0, $t0, $t1   #  the sum is in $t0
sw    $t0, 12($s4)   #  $t0 is stored into d[3]
```

Recap – Numeric Representations

- Decimal 35_{10}
- Binary 00100011_2
- Hexadecimal (compact representation)

  0x 23 or 23_{hex}

  0-15 (decimal) → 0-9, a-f (hex)

Instruction Formats

Instructions are represented as 32-bit numbers (one word), broken into 6 fields

R-type instruction add $t0, $s1, $s2

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct
opcode	source	source	dest	shift amt	function

I-type instruction lw $t0, 32($s3)

6 bits	5 bits	5 bits	16 bits
opcode	rs	rd	constant

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 2.6 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).

Logical Operations

Logical ops	C operators	Java operators	MIPS instr
• Shift Left	<<	<<	sll
• Shift Right	>>	>>>	srl
• Bit-by-bit AND	&	&	and, andi
• Bit-by-bit OR	|	|	or, ori
• Bit-by-bit NOT	~	~	nor

Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2: `beq register1, register2, L1`
  Similarly, bne and slt (set-on-less-than)
- Unconditional branch:

```
j   L1
jr  $s0
```

Convert to assembly:

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

Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2: beq register1, register2, L1
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:
  j L1
  jr $s0

Convert to assembly:

```
if (i == j)                    bne  $s3, $s4, Else
   f = g+h;                    add  $s0, $s1, $s2
else                           j    Exit
   f = g-h;              Else: sub  $s0, $s1, $s2
                         Exit:
```

Example

Convert to assembly:

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

i and k are in $s3 and $s5 and
base of array save[] is in $s6

Example

Convert to assembly:

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

i and k are in $s3 and $s5 and base of array save[] is in $s6

```
Loop:  sll    $t1, $s3, 2
       add    $t1, $t1, $s6
       lw     $t0, 0($t1)
       bne    $t0, $s5, Exit
       addi   $s3, $s3, 1
       j      Loop
Exit:
```

Procedures

- Each procedure (function, subroutine) maintains register values
- When another procedure is called (the callee), the new procedure takes over the registers.
- Values may have to be saved so we can safely return to the caller.

- Seven steps to follow while calling procedures
  1. Place parameters in a place where the procedure can see them.
  2. Transfer control to the procedure.
  3. Acquire the storage resources for the procedure.
  4. Execute the procedure
  5. Place the result value where caller can access it
  6. Release the resources acquired for the procedure
  7. Return control to caller

More Registers in MIPS

- The 32 MIPS registers are partitioned as follows:
  - Register 0 : $zero always stores the constant 0
  - Regs 2-3 : $v0, $v1 return values of a procedure
  - Regs 4-7 : $a0-$a3 input arguments to a procedure
  - Regs 8-15 : $t0-$t7 temporaries
  - Regs 16-23: $s0-$s7 variables
  - Regs 24-25: $t8-$t9 more temporaries
  - Reg 28 : $gp global pointer
  - Reg 29 : $sp stack pointer
  - Reg 30 : $fp frame pointer
  - Reg 31 : $ra return address

Jump-and-Link

- A special register (storage not part of the register file) maintains the address of the **instruction currently being executed** – this is the *program counter* (PC)

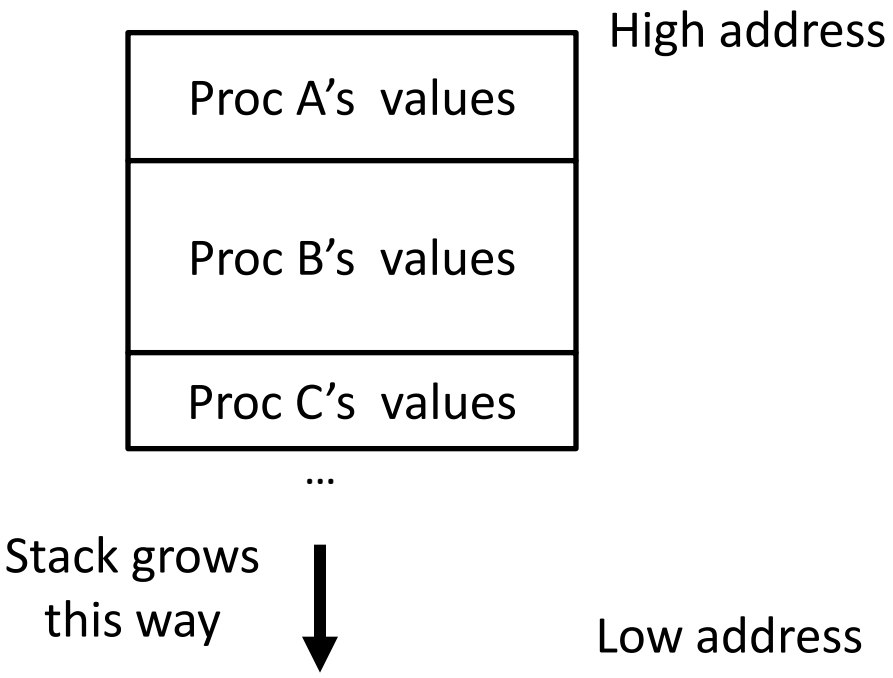
- The procedure call is executed by invoking the jump-and-link (jal) instruction

1. the current PC (actually, PC+4) is saved in the register $ra
2. jump to the procedure's address (the PC is accordingly set to this address)

```
jal    NewProcedureAddress
```

The Stack

- The registers for a procedure seems volatile
  - It seems to disappear every time we switch procedures
  - A procedure's values are therefore backed up in memory on a stack.




```
Proc  A

        call  Proc B
              …
              call Proc C
                 …
              return
        return
return
```

The Stack - II

- Stack grows from higher values to lower values.
- Push – Placing data on the stack. The stack pointer is decremented.
- Pop – Removing data from the stack. The stack pointer is incremented.

Example 1

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

Example 1: A Leaf Procedure

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

Notes:
In this example, the procedure's stack space was used for the caller's variables, not the callee's

The caller took care of saving its $ra and $a0-$a3.

```
leaf_example:
   addi    $sp,  $sp,  -12
   sw      $t1, 8($sp)
   sw      $t0, 4($sp)
   sw      $s0, 0($sp)
   add     $t0, $a0, $a1
   add     $t1, $a2, $a3
   sub     $s0, $t0, $t1
   add     $v0, $s0, $zero
   lw      $s0, 0($sp)
   lw      $t0, 4($sp)
   lw      $t1, 8($sp)
   addi    $sp, $sp, 12
   jr      $ra
```

Saving Registers

- $t0-$t9: 10 temporary registers that are not preserved on a procedure called
- $s0-$s7: 8 saved registers that must be preserved on a procedure call
- Therefore, in the example above there is no need the save the temporary registers. They are only used for values that are never used again.

Example 2

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

Example 2

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

Notes:
The caller saves $a0 and $ra
in its stack space.
Temps are never saved.

```
fact:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
    addi  $v0, $zero, 1
    addi  $sp, $sp, 8
    jr      $ra
L1:
    addi    $a0, $a0, -1
    jal     fact
    lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

Pseudo Instructions

Assemblers allow programmers to use pseudo-instructions like **blt**, which it then translates to two or more instructions

Pseudoinstruction	Translation
bge $rt, $rs, LABEL	slt $t0, $rt, $rs beq $t0, $zero, LABEL
bgt $rt, $rs, LABEL	slt $t0, $rs, $rt bne $t0, $zero, LABEL
ble $rt, $rs, LABEL	slt $t0, $rs, $rt beq $t0, $zero, LABEL
blt $rt, $rs, LABEL	slt $t0, $rt, $rs bne $t0, $zero, LABEL

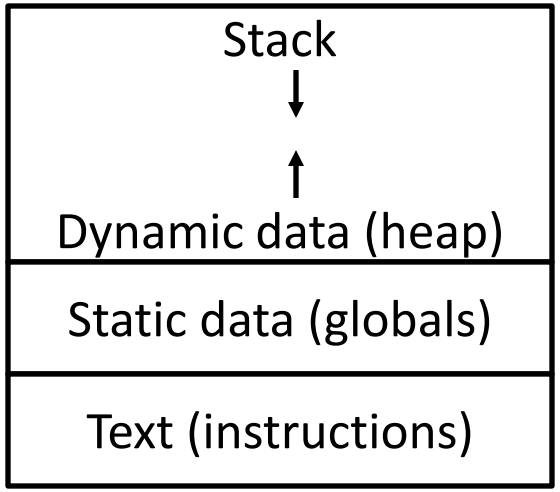
Saves on Stack

- Caller saved
  - $a0-a3 -- old arguments must be saved before setting new arguments for the callee
  - $ra -- must be saved before the jal instruction over-writes this value
  - $t0-t9 -- if you plan to use your temps after the return, save them
    note that callees are free to use temps as they please
  - ✗ You need not save $s0-s7 as the callee will take care of them

- Callee saved
  - $s0-s7 -- before the callee uses such a register, it must save the old contents since the caller will usually need it on return
  - local variables -- space is also created on the stack for variables local to that procedure

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



Instructions Format

- *R-type instruction* add $t0, $s1, $s2

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct
opcode	source	source	dest	shift amt	function

- *I-type instruction* lw $t0, 32($s3)

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	constant

- J-Type instruction jump instruction

6 bits	26 bits
opcode	constant

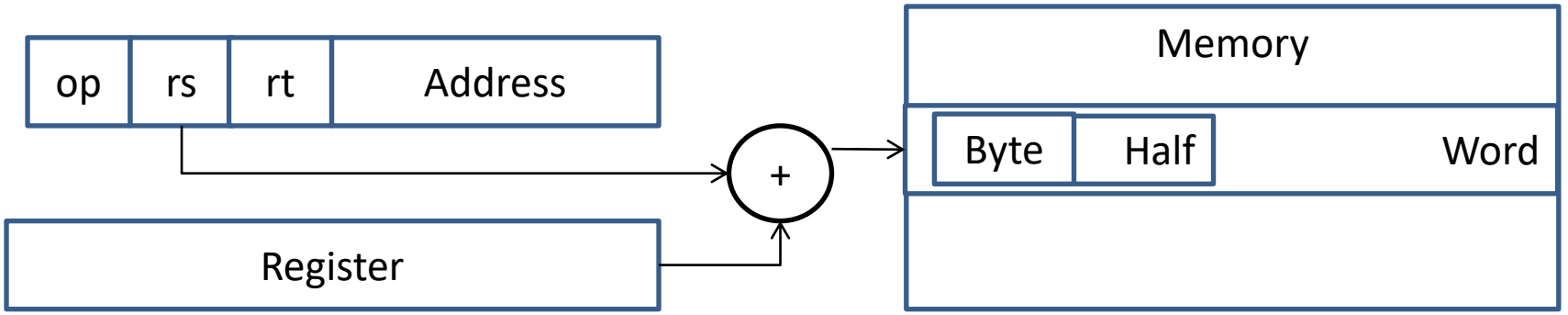
MIPS Addressing Modes Summary

1. Register addressing: operand is a register

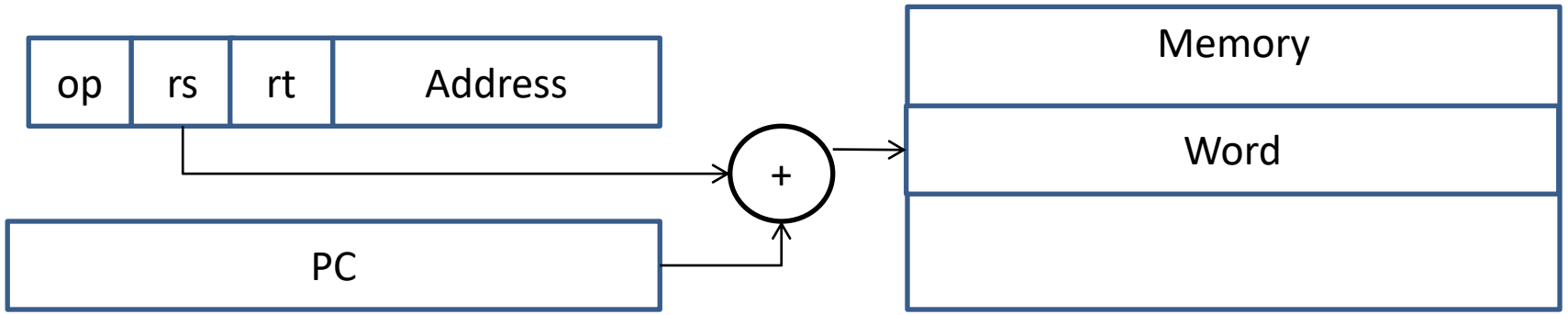
2. Immediate addressing: operand is a constant within the instruction itself

MIPS Addressing Modes Summary

3. Base addressing: where the operand is at the memory location whose address is the sum of a register and a constant



4. PC-relative addressing: where the address is the sum of the PC and a constant in the instruction



MIPS Addressing Modes Summary

5. Pseudodirect addressing: where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC.