CSCI 564 Advanced Computer Architecture

Lecture 04: Introduction to MIPS

Dr. Bo Wu

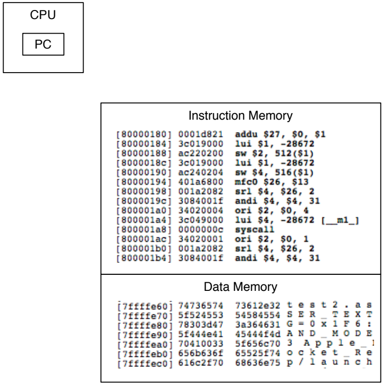
March 3, 2021

Colorado School of Mines

The Idea of the CPU

The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program

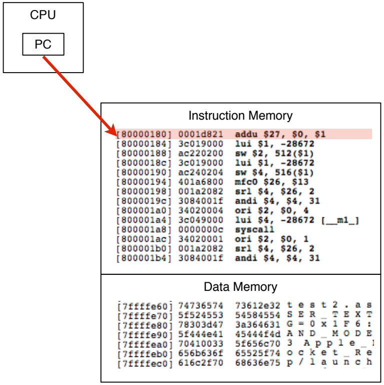


```
[80000180] 0001d821  addu $27, $0, $1
[80000184] 3c019000  lui $1, -28672
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13
[80000198] 001a2082  srl $4, $26, 2
[8000019c] 3084001f  andi $4, $4, 31
[800001a0] 34020004  ori $2, $0, 4
[800001a4] 3c049000  lui $4, -28672 [__m1_]
[800001a8] 0000000c  syscall
[800001ac] 34020001  ori $2, $0, 1
[800001b0] 001a2082  srl $4, $26, 2
[800001b4] 3084001f  andi $4, $4, 31
```

```
[7ffffe60] 74736574  73612e32 t e s t 2 . a s
[7ffffe70] 5f524553  54584554 S E R _ T E X T
[7ffffe80] 78303d47  3a364631 G = 0 x 1 F 6 :
[7ffffe90] 5f444e41  45444f4d A N D _ M O D E
[7ffffea0] 70410033  5f656c70 3   A p p l e _
[7ffffeb0] 656b636f  65525f74 o c k e t _ R e
[7ffffec0] 616c2f70  68636e75 p / l a u n c h
```

The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program

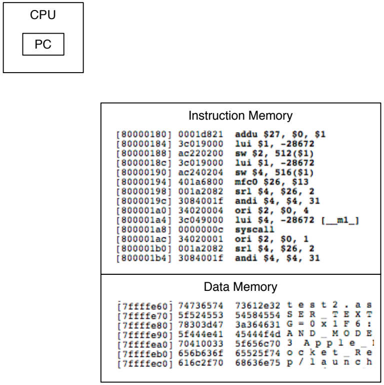


```
[80000180] 0001d821  addu $27, $0, $1
[80000184] 3c019000  lui $1, -28672
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13
[80000198] 001a2082  srl $4, $26, 2
[8000019c] 3084001f  andi $4, $4, 31
[800001a0] 34020004  ori $2, $0, 4
[800001a4] 3c049000  lui $4, -28672 [__m1_]
[800001a8] 0000000c  syscall
[800001ac] 34020001  ori $2, $0, 1
[800001b0] 001a2082  srl $4, $26, 2
[800001b4] 3084001f  andi $4, $4, 31
```

```
[7ffffe60] 74736574  73612e32 t e s t 2 . a s
[7ffffe70] 5f524553  54584554 S E R _ T E X T
[7ffffe80] 78303d47  3a364631 G = 0 x 1 F 6 :
[7ffffe90] 5f444e41  45444f4d A N D _ M O D E
[7ffffea0] 70410033  5f656c70 3   A p p l e _ l
[7ffffeb0] 656b636f  65525f74 o c k e t _ R e
[7ffffec0] 616c2f70  68636e75 p / l a u n c h
```

The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program

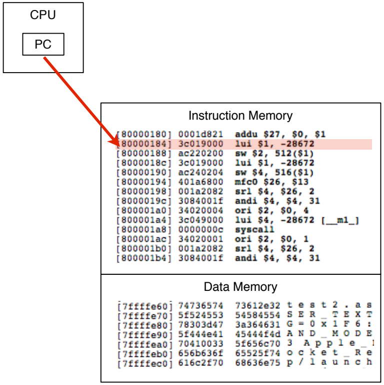


```
[80000180] 0001d821  addu $27, $0, $1
[80000184] 3c019000  lui $1, -28672
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13
[80000198] 001a2082  srl $4, $26, 2
[8000019c] 3084001f  andi $4, $4, 31
[800001a0] 34020004  ori $2, $0, 4
[800001a4] 3c049000  lui $4, -28672 [__m1_]
[800001a8] 0000000c  syscall
[800001ac] 34020001  ori $2, $0, 1
[800001b0] 001a2082  srl $4, $26, 2
[800001b4] 3084001f  andi $4, $4, 31
```

```
[7ffffe60] 74736574  73612e32 t e s t 2 . a s
[7ffffe70] 5f524553  54584554 S E R _ T E X T
[7ffffe80] 78303d47  3a364631 G = 0 x 1 F 6 :
[7ffffe90] 5f444e41  45444f4d A N D _ M O D E
[7ffffea0] 70410033  5f656c70 3   A p p l e _
[7ffffeb0] 656b636f  65525f74 o c k e t _ R e
[7ffffec0] 616c2f70  68636e75 p / l a u n c h
```

The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program

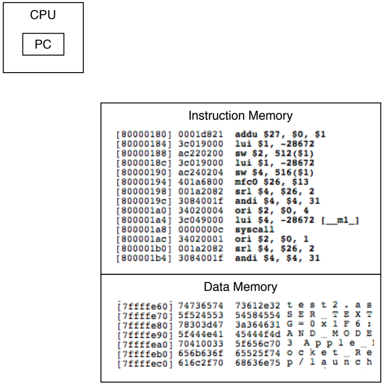


```
[80000180] 0001d821  addu $27, $0, $1
[80000184] 3c019000  lui $1, -28672
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13
[80000198] 001a2082  srl $4, $26, 2
[8000019c] 3084001f  andi $4, $4, 31
[800001a0] 34020004  ori $2, $0, 4
[800001a4] 3c049000  lui $4, -28672 [__m1_]
[800001a8] 0000000c  syscall
[800001ac] 34020001  ori $2, $0, 1
[800001b0] 001a2082  srl $4, $26, 2
[800001b4] 3084001f  andi $4, $4, 31
```

```
[7ffffe60] 74736574  73612e32 t e s t 2 . a s
[7ffffe70] 5f524553  54584554 S E R _ T E X T
[7ffffe80] 78303d47  3a364631 G = 0 x 1 F 6 :
[7ffffe90] 5f444e41  45444f4d A N D _ M O D E
[7ffffea0] 70410033  5f656c70 3   A p p l e _
[7ffffeb0] 656b636f  65525f74 o c k e t _ R e
[7ffffec0] 616c2f70  68636e75 p / l a u n c h
```

The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program

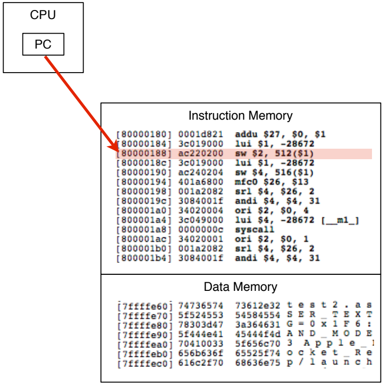


```
[80000180] 0001d821  addu $27, $0, $1
[80000184] 3c019000  lui $1, -28672
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13
[80000198] 001a2082  srl $4, $26, 2
[8000019c] 3084001f  andi $4, $4, 31
[800001a0] 34020004  ori $2, $0, 4
[800001a4] 3c049000  lui $4, -28672 [__m1_]
[800001a8] 0000000c  syscall
[800001ac] 34020001  ori $2, $0, 1
[800001b0] 001a2082  srl $4, $26, 2
[800001b4] 3084001f  andi $4, $4, 31
```

```
[7ffffe60] 74736574  73612e32 t e s t 2 . a s
[7ffffe70] 5f524553  54584554 S E R _ T E X T
[7ffffe80] 78303d47  3a364631 G = 0 x 1 F 6 :
[7ffffe90] 5f444e41  45444f4d A N D _ M O D E
[7ffffea0] 70410033  5f656c70 3   A p p l e _ 
[7ffffeb0] 656b636f  65525f74 o c k e t _ R e
[7ffffec0] 616c2f70  68636e75 p / l a u n c h
```

The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program

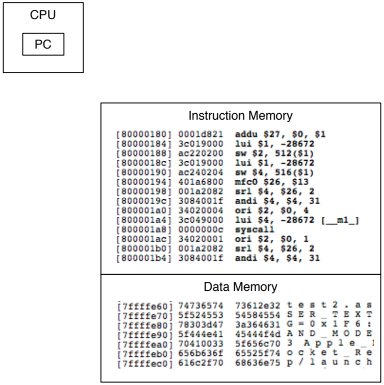


```
[80000180] 0001d821  addu $27, $0, $1
[80000184] 3c019000  lui $1, -28672
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13
[80000198] 001a2082  srl $4, $26, 2
[8000019c] 3084001f  andi $4, $4, 31
[800001a0] 34020004  ori $2, $0, 4
[800001a4] 3c049000  lui $4, -28672 [__m1_]
[800001a8] 0000000c  syscall
[800001ac] 34020001  ori $2, $0, 1
[800001b0] 001a2082  srl $4, $26, 2
[800001b4] 3084001f  andi $4, $4, 31
```

```
[7ffffe60] 74736574  73612e32 t e s t 2 . a s
[7ffffe70] 5f524553  54584554 S E R _ T E X T
[7ffffe80] 78303d47  3a364631 G = 0 x 1 F 6 :
[7ffffe90] 5f444e41  45444f4d A N D _ M O D E
[7ffffea0] 70410033  5f656c70 3   A p p l e _
[7ffffeb0] 656b636f  65525f74 o c k e t _ R e
[7ffffec0] 616c2f70  68636e75 p / l a u n c h
```

The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program

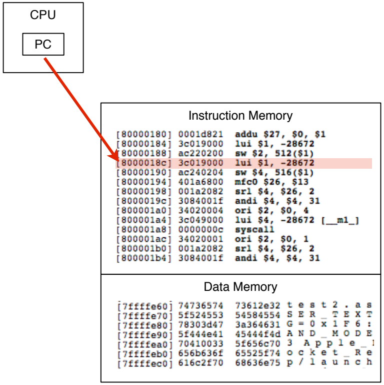


```
[80000180] 0001d821  addu $27, $0, $1
[80000184] 3c019000  lui $1, -28672
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13
[80000198] 001a2082  srl $4, $26, 2
[8000019c] 3084001f  andi $4, $4, 31
[800001a0] 34020004  ori $2, $0, 4
[800001a4] 3c049000  lui $4, -28672 [__m1_]
[800001a8] 0000000c  syscall
[800001ac] 34020001  ori $2, $0, 1
[800001b0] 001a2082  srl $4, $26, 2
[800001b4] 3084001f  andi $4, $4, 31
```

```
[7ffffe60] 74736574  73612e32 t e s t 2 . a s
[7ffffe70] 5f524553  54584554 S E R _ T E X T
[7ffffe80] 78303d47  3a364631 G = 0 x 1 F 6 :
[7ffffe90] 5f444e41  45444f4d A N D _ M O D E
[7ffffea0] 70410033  5f656c70 3   A p p l e _ l
[7ffffeb0] 656b636f  65525f74 o c k e t _ R e
[7ffffec0] 616c2f70  68636e75 p / l a u n c h
```

The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program

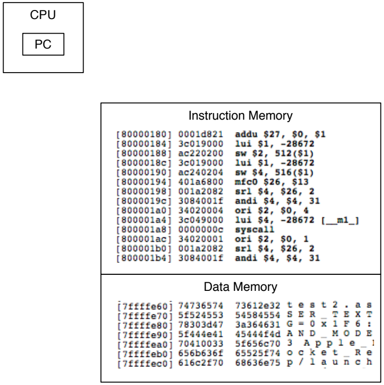


```
[80000180] 0001d821  addu $27, $0, $1
[80000184] 3c019000  lui $1, -28672
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13
[80000198] 001a2082  srl $4, $26, 2
[8000019c] 3084001f  andi $4, $4, 31
[800001a0] 34020004  ori $2, $0, 4
[800001a4] 3c049000  lui $4, -28672 [__m1_]
[800001a8] 0000000c  syscall
[800001ac] 34020001  ori $2, $0, 1
[800001b0] 001a2082  srl $4, $26, 2
[800001b4] 3084001f  andi $4, $4, 31
```

```
[7ffffe60] 74736574  73612e32 t e s t 2 . a s
[7ffffe70] 5f524553  54584554 S E R _ T E X T
[7ffffe80] 78303d47  3a364631 G = 0 x 1 F 6 :
[7ffffe90] 5f444e41  45444f4d A N D _ M O D E
[7ffffea0] 70410033  5f656c70 3   A p p l e _
[7ffffeb0] 656b636f  65525f74 o c k e t _ R e
[7ffffec0] 616c2f70  68636e75 p / l a u n c h
```

The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program



```
[80000180] 0001d821  addu $27, $0, $1
[80000184] 3c019000  lui $1, -28672
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13
[80000198] 001a2082  srl $4, $26, 2
[8000019c] 3084001f  andi $4, $4, 31
[800001a0] 34020004  ori $2, $0, 4
[800001a4] 3c049000  lui $4, -28672 [__m1_]
[800001a8] 0000000c  syscall
[800001ac] 34020001  ori $2, $0, 1
[800001b0] 001a2082  srl $4, $26, 2
[800001b4] 3084001f  andi $4, $4, 31
```

```
[7ffffe60] 74736574  73612e32 t e s t 2 . a s
[7ffffe70] 5f524553  54584554 S E R _ T E X T
[7ffffe80] 78303d47  3a364631 G = 0 x 1 F 6 :
[7ffffe90] 5f444e41  45444f4d A N D _ M O D E
[7ffffea0] 70410033  5f656c70 3   A p p l e _ 
[7ffffeb0] 656b636f  65525f74 o c k e t _ R e
[7ffffec0] 616c2f70  68636e75 p / l a u n c h
```

The Instruction Set Architecture (ISA)

- The ISA is the set of instructions a computer can execute
- All programs are combinations of these instructions

The Instruction Set Architecture (ISA)

- The ISA is the set of instructions a computer can execute
- All programs are combinations of these instructions
- It is an abstraction that programmers (and compilers) use to express computations
  - The ISA defines a set of operations, their semantics, and rules for their use.
  - The software agrees to follow these rules.

The Instruction Set Architecture (ISA)

- The ISA is the set of instructions a computer can execute
- All programs are combinations of these instructions
- It is an abstraction that programmers (and compilers) use to express computations
    - The ISA defines a set of operations, their semantics, and rules for their use.
    - The software agrees to follow these rules.
- The hardware can implement those rules IN ANY WAY IT CHOOSES!
    - Directly in hardware
    - Via a software layer (i.e., a virtual machine)
    - Via a trained monkey with a pen and paper
    - Via a software simulator (like SPIM)

The MIPS ISA

Two ISAs

- MIPS
  - Simple, elegant, easy to implement
  - Designed with the benefit many years ISA design experience
  - Designed for modern programmers, tools, and applications

Two ISAs

- MIPS
  - Simple, elegant, easy to implement
  - Designed with the benefit many years ISA design experience
  - Designed for modern programmers, tools, and applications
- x86
  - Ugly, messy, inelegant, crufty, arcane, very difficult to implement.
  - Designed for 1970s technology
  - Nearly the last in long series of unfortunate ISA designs.
  - The dominant ISA in modern computer systems.

MIPS Basics

- Instructions
  - 4 bytes (32 bits)
  - 4-byte aligned (i.e., they start at addresses that are a multiple of 4 -- 0x0000, 0x0004, etc.)
  - Instructions operate on memory and registers

MIPS Basics

- Instructions
  - 4 bytes (32 bits)
  - 4-byte aligned (i.e., they start at addresses that are a multiple of 4 -- 0x0000, 0x0004, etc.)
  - Instructions operate on memory and registers
- Memory Data types (also aligned)
  - Bytes -- 8 bits
  - Half words -- 16 bits
  - Words -- 32 bits
  - Memory is denote “M” (e.g., M[0x10] is the byte at address 0x10)

MIPS Basics

- Instructions
  - 4 bytes (32 bits)
  - 4-byte aligned (i.e., they start at addresses that are a multiple of 4 -- 0x0000, 0x0004, etc.)
  - Instructions operate on memory and registers
- Memory Data types (also aligned)
  - Bytes -- 8 bits
  - Half words -- 16 bits
  - Words -- 32 bits
  - Memory is denote “M” (e.g., M[0x10] is the byte at address 0x10)
- Registers
  - 32 4-byte registers in the “register file”
  - Denoted “R” (e.g., R[2] is register 2)

Bytes and Words

Byte addresses

Address	Data
0x0000	0xAA
0x0001	0x15
0x0002	0x13
0x0003	0xFF
0x0004	0x76
...	.
0xFFFE	.
0xFFFF	.

Half Word Addrs

Address	Data
0x0000	0xAA15
0x0002	0x13FF
0x0004	.
0x0006	.
...	.
...	.
...	.
0xFFFC	.

Word Addresses

Address	Data
0x0000	0xAA1513FF
0x0004	.
0x0008	.
0x000C	.
...	.
...	.
...	.
0xFFFC	.

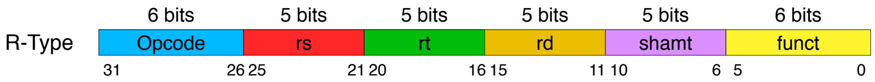
- In modern ISAs (including MIPS) memory is “byte addressable”
- In MIPS, half words and words are aligned.

The MIPS Register File

- All registers are the same
  - Where a register is needed any register will work
- By convention, we use them for particular tasks
  - Argument passing
  - Temporaries, etc.
  - These rules (“the register discipline”) are part of the ISA
- $zero is the “zero register”
  - It is always zero.
  - Writes to it have no effect.

Name	number	use	Callee saved
$zero	0	zero	n/a
$at	1	Assemble Temp	no
$v0 - $v1	2 - 3	return value	no
$a0 - $a3	4 - 7	arguments	no
$t0 - $t7	8 - 15	temporaries	no
$s0 - $s7	16 - 23	saved temporaries	yes
$t8 - $t9	24 - 25	temporaries	no
$k0 - $k1	26 - 27	Res. for OS	yes
$gp	28	global ptr	yes
$sp	29	stack ptr	yes
$fp	30	frame ptr	yes
$ra	31	return address	yes

MIPS R-Type Arithmetic Instructions



- R-Type instructions encode operations of the form "a = b OP c" where 'OP' is +, -, <<, &, etc.
  - More formally, R[rd] = R[rs] OP R[rt]
- Bit fields
  - "opcode" encodes the operation type.
  - "funct" specifies the particular operation.
  - "rs" are "rt" source registers; "rd" is the destination register
    - 5 bits can specify one of 32 registers.
- "shamt" is the "shift amount" for shift operations
  - Since registers are 32 bits, 5 bits are sufficient

Examples

- `add $t0, $t1, $t2`
  - `R[8] = R[9] + R[10]`
  - `opcode = 0, funct = 0x20`
- `nor $a0, $s0, $t4`
  - `R[4] = ~(R[16] | R[12])`
  - `opcode = 0, funct = 0x27`
- `sll $t0, $t1, 4`
  - `R[4] = R[16] << 4`
  - `opcode = 0, funct = 0x0, shamt = 4`

MIPS R-Type Control Instructions

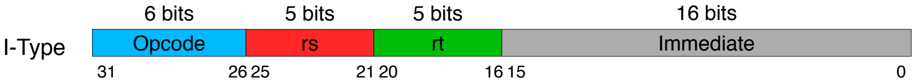
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R-Type	Opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0

- R-Type encodes “register-indirect” jumps
- Jump register
  - jr rs: PC = R[rs]
- Jump and link register
  - jalr rs, rd: R[rd] = PC + 8; PC = R[rs]
  - rd default to $ra (i.e., the assembler will fill it in if you leave it out)

Examples

- `jr $t2`
  - `PC = r[10]`
  - `opcode = 0, funct = 0x8`
- `jalr $t0`
  - `PC = R[8]`
  - `R[31] = PC + 8`
  - `opcode = 0, funct = 0x9`

MIPS I-Type Arithmetic Instructions

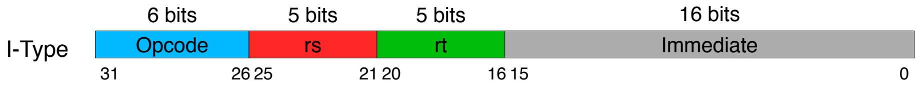


- I-Type arithmetic instructions encode operations of the form “a = b OP #”
- ‘OP’ is +, -, <<, &, etc and # is an integer constant
  - More formally, e.g.: R[rd] = R[rs] + 42
- Components
  - “opcode” encodes the operation type.
  - “rs” is the source register
  - “rd” is the destination register
- “immediate” is a 16 bit constant used as an argument for the operation

Examples

- `addi $t0, $t1, -42`
  - `R[8] = R[9] + -42`
  - `opcode = 0x8`
- `ori $t0, $zero, 42`
  - `R[4] = R[0] | 42`
  - `opcode = 0xd`
  - `Loads a constant into $t0`

MIPS I-Type Branch Instructions

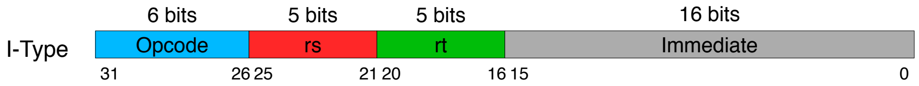


- I-Type also encode branches
  - if (R[rd] OP R[rs])
    PC = PC + 4 + 4 * Immediate
    else
    PC = PC + 4
- Components
  - "rs" and "rt" are the two registers to be compared
  - "rt" is sometimes used to specify branch type.
- "immediate" is a 16 bit branch offset
  - It is the signed offset to the target of the branch
  - Limits branch distance to 32K instructions
  - Usually specified as a label, and the assembler fills it in for you.

Examples

- `beq $t0, $t1, -42`
  - `if R[8] == R[9]`
    `PC = PC + 4 + 4*-42`
  - `opcode = 0x4`
- `bgez $t0, -42`
  - `if R[8] >= 0`
    `PC = PC + 4 + 4*-42`
  - `opcode = 0x1`
  - `rt = 1`

MIPS I-Type Memory Instructions

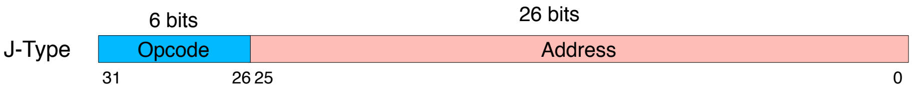


- I-Type also encode memory access
  - Store: M[R[rs] + Immediate] = R[rt]
  - Load: R[rt] = M[R[rs] + Immediate]
- MIPS has load/stores for byte, half word, and word
- Sub-word loads can also be signed or unsigned
  - Signed loads sign-extend the value to fill a 32 bit register.
  - Unsigned zero-extend the value.
- “immediate” is a 16 bit offset
  - Useful for accessing structure components
  - It is signed.

Examples

- `lw $t0, 4($t1)`
  - `R[8] = M[R[9] + 4]`
  - opcode = `0x23`
- `sb $t0, -17($t1)`
  - `M[R[12] + -17] = R[4]`
  - opcode = `0x28`

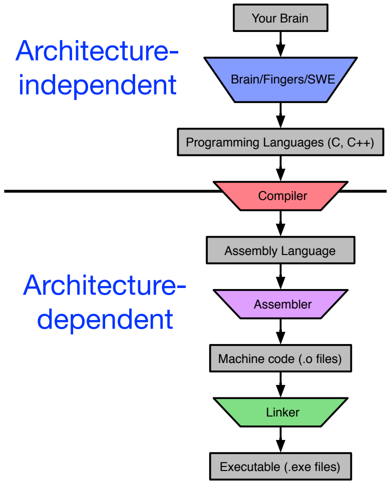
MIPS J-Type Instructions



- J-Type encodes the jump instructions
- Plain Jump
  - JumpAddress = {PC+4[31:28],Address,2’b0}
  - Address replaces *most* of the PC
  - PC = JumpAddress
- Jump and Link
  - R[$ra] = PC + 8; PC = JumpAddress;
- J-Type also encodes misc instructions
  - syscall, interrupt return, and break

From C to MIPS

Compiling: C to bits

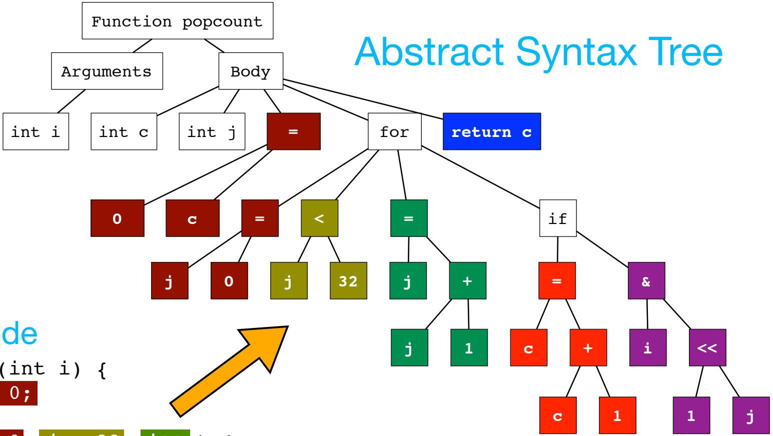


C Code

Count the number of 1's in the binary representation of i

```
int popcount(int i) {
     int c = 0;
     int j;
     for(j = 0; j < 32; j++ ) {
      if (i & (1 << j))
           c++;
     }
     return c;
}
```

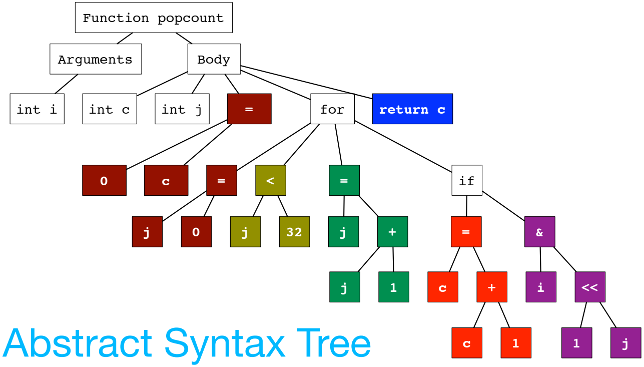
In the Compiler



C-Code

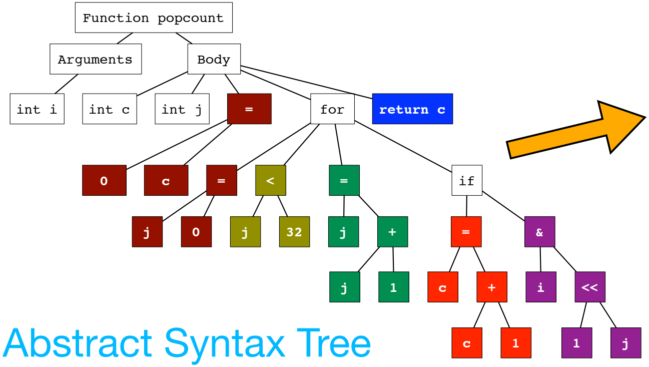
```
int popcount(int i) {
     int c = 0;
     int j;
     for(j = 0; j < 32; j++ ) {
      if (i & (1 << j))
          c++;
     }
     return c;
}
```

In the Compiler

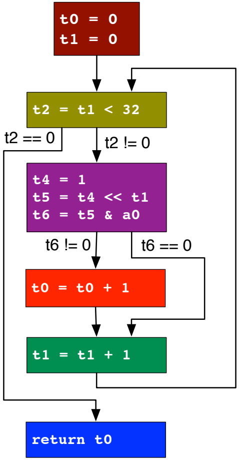


Abstract Syntax Tree

In the Compiler

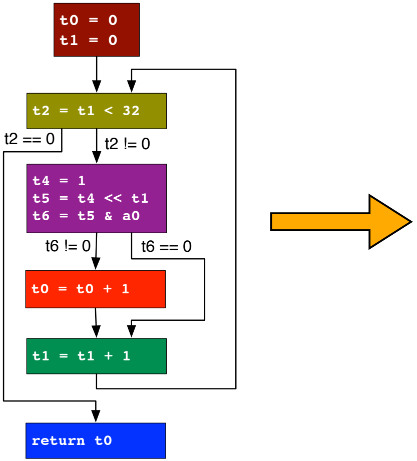


Abstract Syntax Tree



Control Flow Graph

In the Compiler



```
popcount:
    ori $v0, $zero, 0
    ori $t1, $zero, 0
top:
    slti $t2, $t1, 32
    beq  $t2, $zero, end
    nop
    addi $t3, $zero, 1
    sllv $t3, $t3, $t1
    and $t3, $a0, $t3
    beq $t3, $zero, notone
    nop
    addi $v0, $v0, 1
notone:
    beq $zero, $zero, top
    addi $t1, $t1, 1
end:
    jr $ra
    nop
```

Control flow graph

Assembly

In the Assembler

```
popcount:
    ori $v0, $zero, 0
    ori $t1, $zero, 0
top:
    slti $t2, $t1, 32
    beq  $t2, $zero, end
    nop
    addi $t3, $zero, 1
    sllv $t3, $t3, $t1
    and $t3, $a0, $t3
    beq $t3, $zero, notone
    nop
    addi $v0, $v0, 1
notone:
    beq $zero, $zero, top
    addi $t1, $t1, 1
end:
    jr $ra
    nop
```

→

```
00110100000000100000000000000000
00110100000010010000000000000000
00101001001010100000000000100000
00010001010000000000000000001001
00000000000000000000000000000000
00100000000010110000000000000001
00000001001010110101100000000100
00000000100010110101100000100100
00010001011000000000000000000010
00000000000000000000000000000000
00100000010000100000000000000001
00010000000000001111111111110110
00100001001010010000000000000001
00000011111000000000000000001000
00000000000000000000000000000000
```

Assembly

Executable Binary

In the Compiler

```
int popcount(int i) {
     int c = 0;
     int j;
     for(j = 0; j < 32; j++ ) {
      if (i & (1 << j))
          c++;
     }
     return c;
}
```

C-Code

```
  popcount:
        ori $v0, $zero, 0
        ori $t1, $zero, 0
  top:
        slti $t2, $t1, 32
        beq  $t2, $zero, end
        nop
        addi $t3, $zero, 1
        sllv $t3, $t3, $t1
        and $t3, $a0, $t3
        beq $t3, $zero, notone
        nop
        addi $v0, $v0, 1
  notone:
        beq $zero, $zero, top
        addi $t1, $t1, 1
  end:
        jr $ra
        nop
```

Assembly

Take a look: http://llvm.org/

Top Reasons to Use Assembly Code

Top Reasons to Use Assembly Code

- You are writing a compiler, so you don’t have a choice

Top Reasons to Use Assembly Code

- You are writing a compiler, so you don’t have a choice
- You want to understand what the machine is doing

Top Reasons to Use Assembly Code

- You are writing a compiler, so you don’t have a choice
- You want to understand what the machine is doing
- Assembly code is probably faster

Top Reasons to Use Assembly Code

- You are writing a compiler, so you don’t have a choice
- You want to understand what the machine is doing
- Assembly code is probably faster
- You want to show other people you are cool