UNT
UNIVERSITY OF NORTH TEXAS

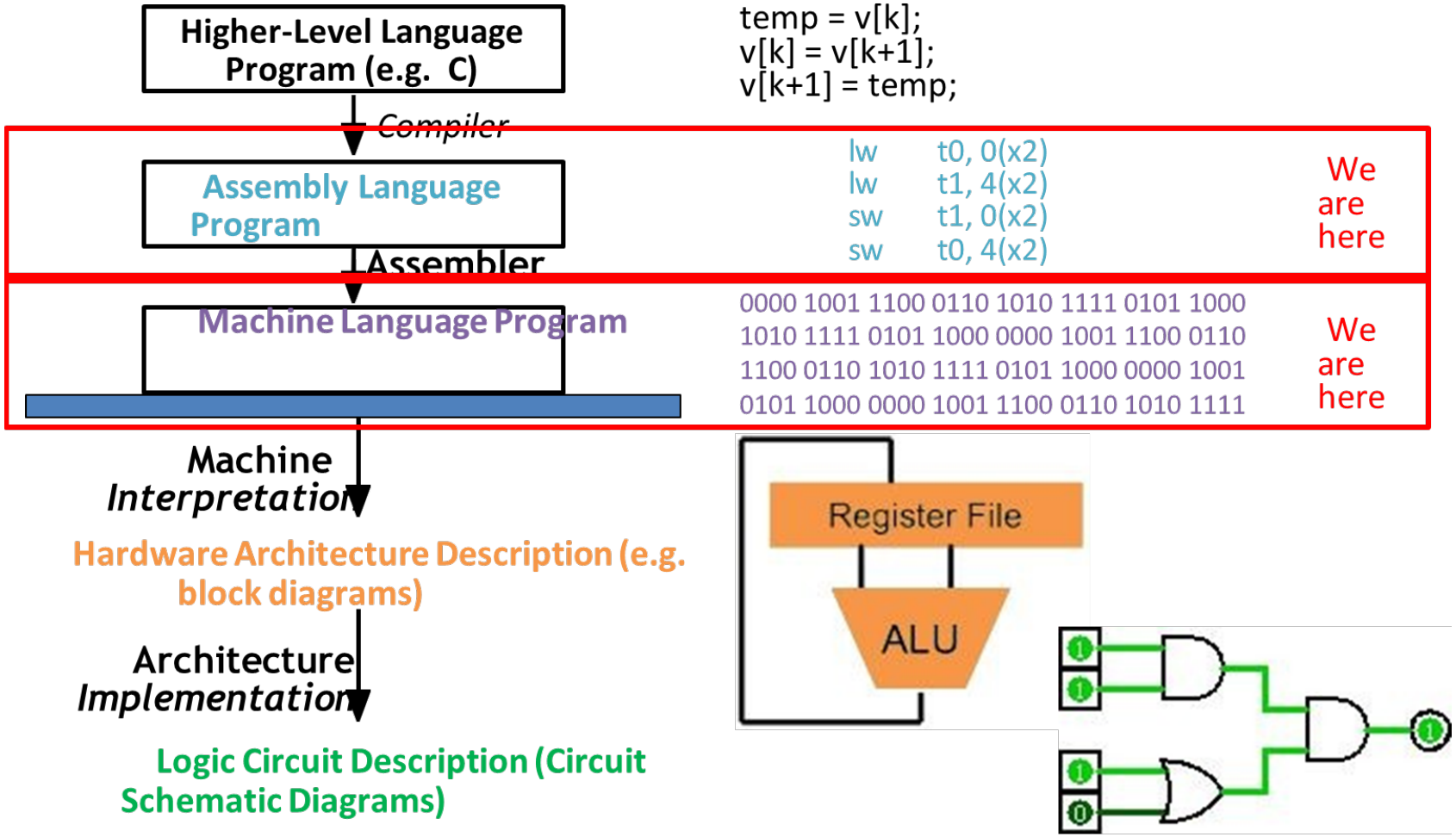
CSCE 5610
Computer System Architecture

Machine Organization and Assembly Language

Content

- Assembly Language
- Machine Language

Great Idea #1: Abstraction



Machine language

- **Machine language**, the binary representation for instructions.
  - — We'll see how it is **designed for the common case**
    - Fixed-sized (32-bit) instructions
    - Only 3 instruction formats
    - Limited-sized immediate fields

Assembly vs. machine language

- So far we've been using assembly language.
  - We assign names to operations (e.g., add) and operands (e.g., $t0).
  - Branches and jumps use labels instead of actual addresses.
  - Assemblers support many pseudo-instructions.

- Programs must eventually be translated into machine language, a binary format that can be stored in memory and decoded by the CPU.

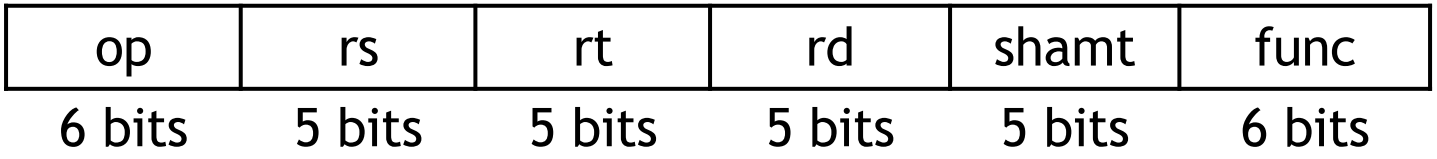
- MIPS machine language is designed to be easy to decode.
  - Each MIPS instruction is the same length, 32 bits.
  - There are only three different instruction formats, which are very similar to each other.

Three instruction formats

- R-type format
- I-type format
- J-type format

R-type format

- Register-to-register arithmetic instructions use the R-type format.



- This format includes six different fields.
  - op is an operation code or opcode that selects a specific operation.
  - rs and rt are the first and second source registers.
  - rd is the destination register.
  - shamt is only used for shift instructions.
  - func is used together with op to select an arithmetic instruction.

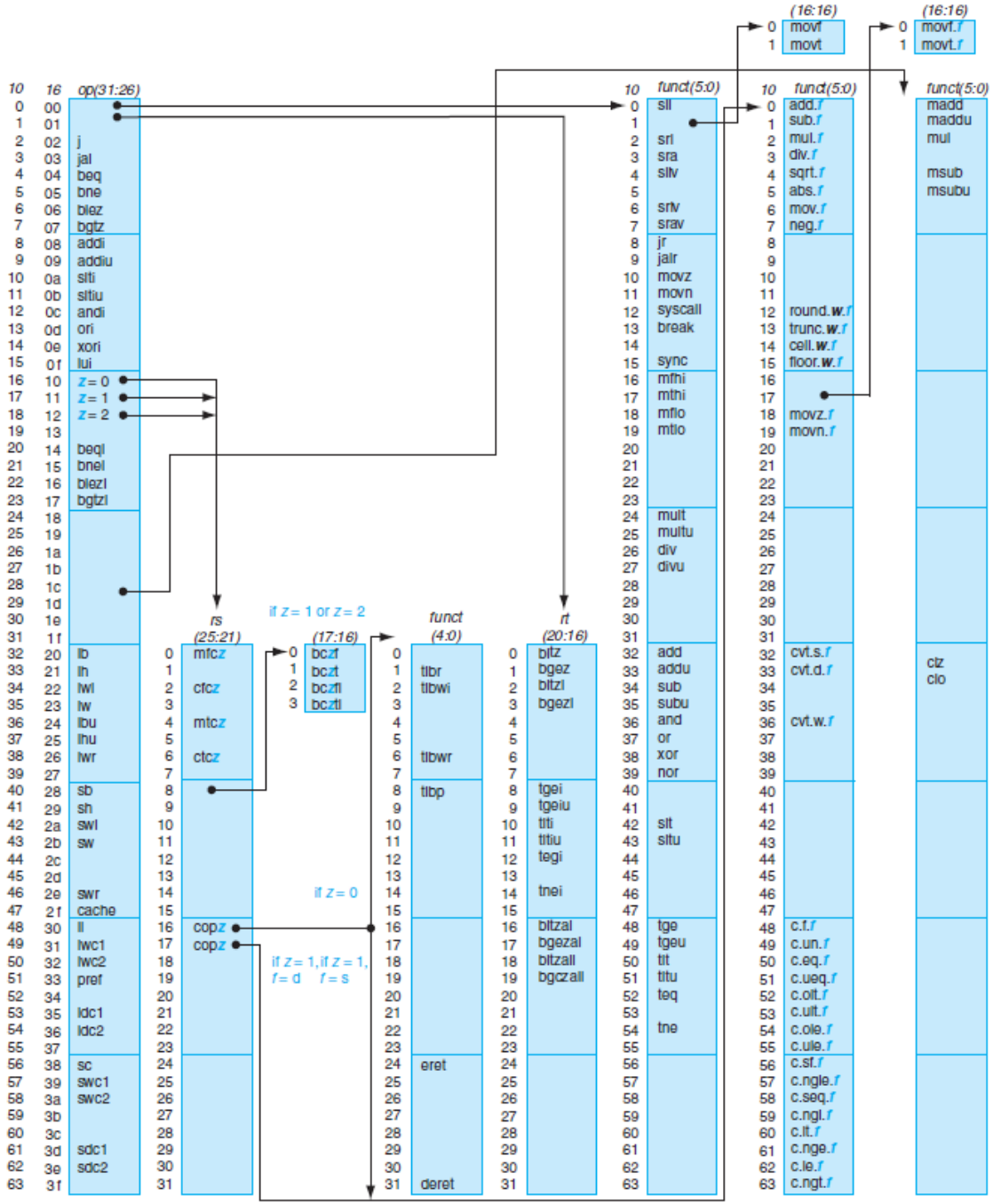
About the registers

- We have to encode register names as 5-bit numbers from 00000 to 11111.
  - For example, $t8 is register $24, which is represented as 11000.

- The number of registers available affects the instruction length.
  - Each R-type instruction references 3 registers, which requires a total of 15 bits in the instruction word.
  - We can't add more registers without either making instructions longer than 32 bits, or shortening other fields like op and possibly reducing the number of available operations.

Register Correspondences

$zero	$0	Zero
$at	$1	Assembler temp
$v0-$v1	$2-3	Value (return from function)
$a0-$a3	$4-7	Argument (to function)
$t0-$t7	$8-15	Temporaries
$s0-$s7	$16-23	Saved Temporaries Saved
$t8-$t9	$24-25	Temporaries
$k0-$k1	$26-27	Kernel (OS) Registers
$gp	$28	Global Pointer Saved
$sp	$29	Stack Pointer Saved
$fp	$30	Frame Pointer Saved
$ra	$31	Return Address Saved


op(31:26)
rs (25:21)
rt (20:16)
funct (4:0)
funct(5:0)
(16:16)
if z = 1 or z = 2
if z = 0
if z = 1, if z = 1, f = d f = s
j
jal
beq
bne
blez
bgtz
addi
addiu
slti
sltiu
andi
ori
xori
lui
z = 0
z = 1
z = 2
beql
bnel
blezl
bgtzl
lb
lh
lwl
lw
lbu
lhu
lwr
sb
sh
swl
sw
swr
cache
ll
lwc1
lwc2
pref
ldc1
ldc2
sc
swc1
swc2
sdc1
sdc2
mfcz
cfcz
mtcz
ctcz
copz
copz
bczf
bczt
bczfl
bcztl
tlbr
tlbwi
tlbwr
tlbp
eret
deret
bltz
bgez
bltzl
bgezl
tgei
tgeiu
tlti
tltiu
teqi
tnei
bltzal
bgezal
bltzall
bgczall
sll
srl
sra
sllv
srlv
srav
jr
jalr
movz
movn
syscall
break
sync
mfhi
mthi
mflo
mtlo
mult
multu
div
divu
add
addu
sub
subu
and
or
xor
nor
slt
sltu
tge
tgeu
tlt
tltu
teq
tne
movf
movt
movf.f
movt.f
add.f
sub.f
mul.f
div.f
sqrt.f
abs.f
mov.f
neg.f
round.w.f
trunc.w.f
ceil.w.f
floor.w.f
movz.f
movn.f
cvt.s.f
cvt.d.f
cvt.w.f
c.f.f
c.un.f
c.eq.f
c.ueq.f
c.olt.f
c.ult.f
c.ole.f
c.ule.f
c.sf.f
c.ngle.f
c.seq.f
c.ngl.f
c.lt.f
c.nge.f
c.le.f
c.ngt.f
madd
maddu
mul
msub
msubu
clz
clo

Exercise

- What should be the corresponding machine code for the following MIPS assembly code:
  - add $t1, $t1, $zero

Breaking Down add $t1, $t1, $zero:

opcode: 000000 (indicating an R-type instruction).

rs: $t1 is the source register for the first operand, which corresponds to register $t1 (register 9). So, rs = 01001.

rt: $zero is the second operand, which corresponds to register $zero (register 0). So, rt = 00000.

rd: The destination register is $t1 (register 9). So, rd = 01001.

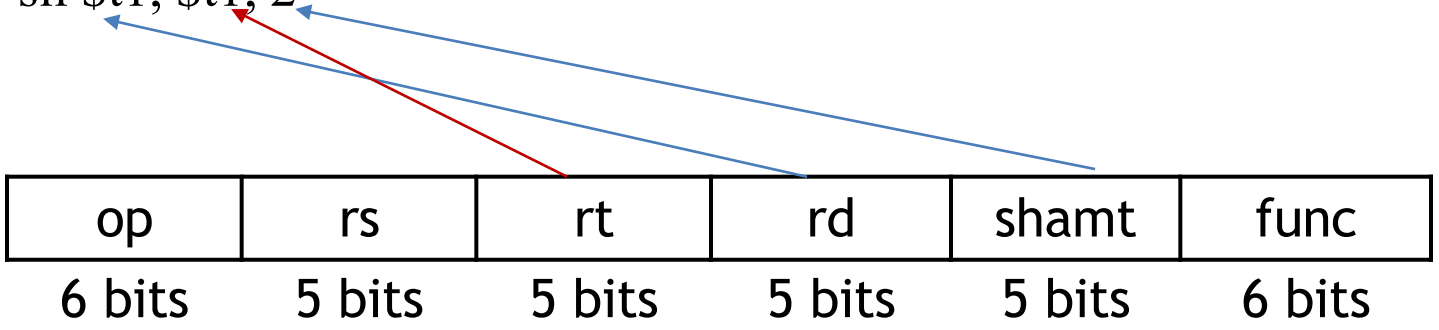
shamt: No shift is being performed, so shamt = 00000.

funct: For the add operation, the funct code is 100000.

opcode | rs | rt | rd | shamt | funct
000000 | 01001 | 00000 | 01001 | 00000 | 100000

Exercise

- What should be the corresponding machine code for the following MIPS assembly code:
  - sll $t1, $t1, 2

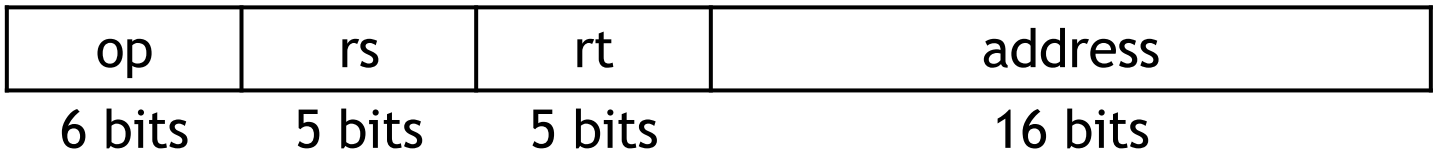


opcode | rs | rt | rd | shamt | funct
000000 | 00000 | 01001 | 01001 | 00010 | 000000

Why rs is Not Used in sll:
The sll instruction only requires two things:
The value to be shifted (which is in the rt register).
The amount to shift (which is provided by the shamt field in the instruction).

I-type format

- Load, store, branch and immediate instructions all use the I-type format.



- For uniformity, op, rs and rt are in the same positions as in the R-format.
- The meaning of the register fields depends on the exact instruction.
  - rs is a source register—an address for loads and stores, or an operand for branch and immediate arithmetic instructions.
  - rt is a source register for branches, but a destination register for the other I-type instructions.
- The address is a 16-bit signed two's-complement value.
  - It can range from -32,768 to +32,767.
  - But that's not always enough!

Exercise

- What should be the corresponding machine code for the following MIPS assembly code:
  - lw $t1, 0($sp)
  - sw $t1, 0($sp)

Where:
opcode: 6 bits for the operation code. For lw, the opcode is 100011.
rs: 5 bits for the base register (the register holding the address).
rt: 5 bits for the target register (where the loaded word will be stored).
immediate: 16 bits for the offset.
Breaking Down lw $t1, 0($sp):
opcode: The opcode for lw is 100011 (binary).
rs: The base register is $sp (stack pointer), which corresponds to register 29. In binary, $sp = 11101.
rt: The target register is $t1, which corresponds to register 9. In binary, $t1 = 01001.
immediate: The offset is 0, so in 16-bit binary, this is 0000000000000000.

opcode | rs | rt | immediate
100011 | 11101 | 01001 | 0000000000000000

Breaking Down lw $t1, 4($sp):
opcode: The opcode for lw is 100011 (binary).
rs: The base register is $sp, which corresponds to register 29. In binary, $sp = 11101.
rt: The target register is $t1, which corresponds to register 9. In binary, $t1 = 01001.
immediate: The offset is 4. The binary representation of 4 in 16 bits is 0000000000000100.
Final Machine Code:
Now, putting everything together:
Copy code
opcode | rs | rt | immediate
100011 | 11101 | 01001 | 0000000000000100

Where:
opcode: 6 bits for the operation code. For sw, the opcode is 101011.
rs: 5 bits for the base register, which holds the base address. In this case, $sp (stack pointer), which corresponds to register 29.
rt: 5 bits for the source register, which contains the value to be stored. In this case, $t1, which corresponds to register 9.
immediate: 16 bits for the offset. In this case, the offset is 0.
Breaking Down sw $t1, 0($sp):
opcode: The opcode for sw is 101011 (binary).
rs: The base register is $sp (stack pointer), which corresponds to register 29. In binary, $sp = 11101.
rt: The source register is $t1, which corresponds to register 9. In binary, $t1 = 01001.
immediate: The offset is 0. The binary representation of 0 in 16 bits is 0000000000000000.
Final Machine Code:
Now, let's put everything together into the 32-bit machine code for sw $t1, 0($sp):

opcode | rs | rt | immediate
101011 | 11101 | 01001 | 0000000000000000

Two Complement

- **Easy to do in HW:**
  - Most significant bit tells sign (sign bit)
  - Addition can be done without anything special

- How?
  - Invert all bits and add one.

0	1	1	1	1	1	1	1	=	127
0	0	0	0	0	0	1	0	=	2
0	0	0	0	0	0	0	1	=	1
0	0	0	0	0	0	0	0	=	0
1	1	1	1	1	1	1	1	=	−1
1	1	1	1	1	1	1	0	=	−2
1	0	0	0	0	0	0	1	=	−127
1	0	0	0	0	0	0	0	=	−128

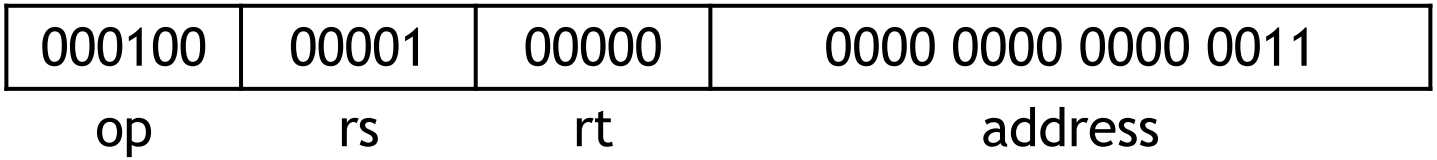
8-bit two's-complement integers

Branches

- For branch instructions, the constant field is not an address, but an *offset* from the current program counter (PC) to the target address.

```
        beq   $at, $0, L
        add   $v1, $v0, $0
        add   $v1, $v1, $v1
        j     Somewhere
L:      add   $v1, $v0, $v0
```

- Since the branch target L is three *instructions* past the beq, the address field would contain 3. The whole beq instruction would be stored as:



Larger branch constants

- Empirical studies of real programs show that most branches go to targets less than 32,767 instructions away—branches are mostly used in loops and conditionals, and programmers are taught to make code bodies short.
- If you do need to branch further, you can use a jump with a branch. For example, if "Far" is very far away, then the effect of:

```
        beq     $s0, $s1, Far
        ...
```

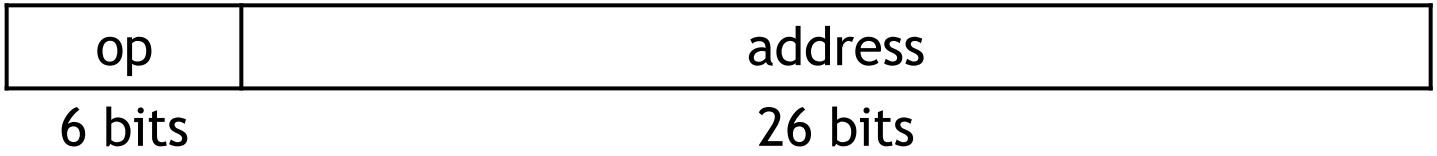
can be simulated with the following actual code.

```
        bne     $s0, $s1, Next
        j       Far
Next:   ...
```

- Again, the MIPS designers have taken care of the common case first.

J-type format

- Finally, the jump instruction uses the J-type instruction format.



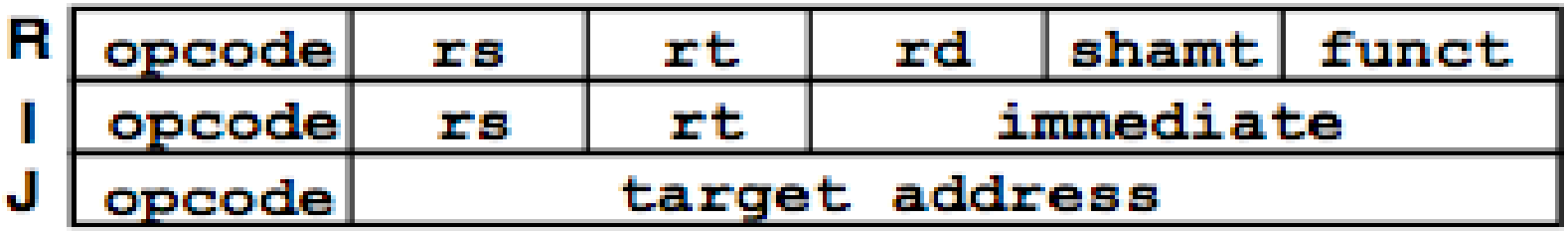
- The jump instruction contains a *word* address, not an offset
  - Remember that each MIPS instruction is one word long, and word addresses must be divisible by four.
  - So instead of saying "jump to address 4000," it's enough to just say "jump to instruction 1000."
  - A 26-bit address field lets you jump to any address from 0 to 2^{28}.

- For even longer jumps, the jump register, or jr, instruction can be used.

```
jr      $ra             # Jump to 32-bit address in register $ra
```

Summary of Machine Language

- Machine language is the binary representation of instructions:
  - The format in which the machine actually executes them
- MIPS machine language is designed to simplify processor implementation
  - Fixed length instructions
  - 3 instruction encodings: R-type, I-type, and J-type
  - Common operations fit in 1 instruction
    - Uncommon (e.g., long immediates) require more than one



Decoding Machine Language

How do we convert 1s and 0s to assembly language and to C code?

Machine language --> assembly --> C?

For each 32 bits:

1. Look at opcode to distinguish between R- Format, J-Format, and I-Format
2. Use instruction format to determine which fields exist
3. Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number
4. Logically convert this MIPS code into valid C code. Always possible? Unique?

Decoding (1/7)

- Here are six machine language instructions in hexadecimal:

  00001025_{hex}

  $0005402A_{hex}$

  11000003_{hex}

  00441020_{hex}

  $20A5FFFF_{hex}$

  08100001_{hex}

- Let the first instruction be at address $4{,}194{,}304_{ten}$ (0x00400000hex)
- Next step: convert hex to binary

Decoding (2/7)

- The six machine language instructions in binary:

00000000000000000100000100101

00000000000010101000000000101010

00010001000000000000000000000011

00000000010001000001000000100000

00100000101001011111111111111111

00001000000100000000000000000001

- Next step: identify opcode and format

R	0	rs	rt	rd	shamt	funct
I	1, 4-62	rs	rt	immediate		
J	2 or 3	target address				

Decoding (3/7)

- Select the opcode (first 6 bits) to determine the format:

000000 00000 00000 00010 00000 100101

000000 00000 00101 01000 00000 101010

000100 01000 00000 00000 00000 000011

000000 00010 00100 00010 00000 100000

001000 00101 00101 11111 11111 111111

000010 00000 10000 00000 00000 000001

- Look at opcode: 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format
- Next step: separation of fields R R I R I J Format:

R	0	rs	rt	rd	shamt	funct
I	1, 4-62	rs	rt	immediate		
J	2 or 3	target address				

Decoding (4/7)

- Fields separated based on format/opcode:

Format:						
R	0	0	0	2	0	37
R	0	0	5	8	0	42
I	4	8	0	+3		
R	0	2	4	2	0	32
I	8	5	5	-1		
J	2	1,048,577				

- Next step: translate (“disassemble”) MIPS assembly instructions R R I R I J Format:

Decoding (5/7)

- MIPS Assembly (Part 1):
- Address: Assembly instructions:

```
0x00400000      or    $2,$0,$0
0x00400004      slt   $8,$0,$5
0x00400008      beq   $8,$0,3
0x0040000c      add   $2,$2,$4
0x00400010      addi  $5,$5,-1
0x00400014      j     0x100001
```

- Better solution: translate to more meaningful MIPS instructions (fix the branch/jump and add labels, registers)

Decoding (6/7)

- MIPS Assembly (Part 2):

```
        or    $v0,$0,$0
Loop:   slt   $t0,$0,$a1
        beq   $t0,$0,Exit
        add   $v0,$v0,$a0
        addi  $a1,$a1,-1
        j     Loop
Exit:
```

- Next step: translate to C code (must be creative!)

Decoding (7/7)

- Possible C code:

```
v0 = 0;
while (a1 > 0) {
   v0 += a0;
   a1 -= 1;
}
```

```
        or    $v0,$0,$0
Loop:   slt   $t0,$0,$a1
        beq   $t0,$0,Exit
        add   $v0,$v0,$a0
        addi  $a1,$a1,-1
        j     Loop
Exit:
```