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:

  00000000000000000001000000100101
  00000000000001010100000000101010
  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:
```

Exercise

```
int pow(int n, int m) {
        int res = 1;
        for(int i = 0; i < m; i++){
                res = res * n;
        }
        return res;
}
```

pow:

```
li $a0, 2; //n = 2
li $a1, 3; //m = 3
jal pow
```

```
main:
   # Set up for the pow function call
   li   $a0, 2        # Load n = 2 into $a0
   li   $a1, 3        # Load m = 3 into $a1
   jal  pow           # Call pow function

# After returning from pow, the result is in $v0
   move $t0, $v0      # Move the result into $t0 (just for exampl

   # Exit program (using syscall)
   li   $v0, 10       # Load 10 into $v0 (exit syscall)
   syscall            # Exit the program

# pow function
pow:
   # Function prologue: Save return address and registers
   addi $sp, $sp, -8   # Make space on stack for saving registe
   sw   $ra, 4($sp)    # Save return address
   sw   $a2, 0($sp)    # Save $a2 (will be used for loop counter

   # Initialize res = 1 (store in $t0)
   li   $t0, 1        # res = 1

   # Initialize i = 0 (store in $t2)
   li   $t2, 0        # i = 0

pow_loop:
   bge  $t2, $a1, pow_end  # if i >= m, exit loop
   mul  $t0, $t0, $a0      # res = res * n (res = res * n)
   addi $t2, $t2, 1        # i = i + 1
   j    pow_loop           # Jump to the start of the loop

pow_end:
   move $v0, $t0      # Store result in $v0 (return value)

   # Function epilogue: Restore saved registers
   lw   $ra, 4($sp)   # Restore return address
   lw   $a2, 0($sp)   # Restore $a2 (loop counter)
   addi $sp, $sp, 8   # Deallocate space on stack

   jr   $ra           # Return to the caller
```

What should be the corresponding machine code for the following MIPS assembly code:

- sll $t1, $t1, 2

op	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

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).

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:
  - 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
```

Address Translation Example #1

Virtual Address: 0x03D4

TLBT | TLBI

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN | VPO

VPN <u>0x0F</u>   TLBI <u>3</u>   TLBT <u>0x03</u>   TLB Hit? <u>Y</u>   Page Fault? <u>N</u>   PPN: <u>0x0D</u>

Physical Address

CT | CI | CO

11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0

PPN | PPO

CO <u>0</u>   CI <u>0x5</u>   CT <u>0x0D</u>   Hit? <u>Y</u>   Byte: <u>0x36</u>