Topics for Chapter 2

- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS
  - Memory access instructions (Chapters 2.3, 2.9)
  - ➔ Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 2:**
  - Procedure Calls (Chapters 2.8)
- **Part 3:**
  - RISC-V Instruction Format (Chapter 2.5)
  - RISC-V Addressing Modes (Chapter 2.10)

Two Different Views of a Register

- **View contents of register as a single quantity (such as a signed or unsigned integer)**
  - Arithmetic (`add`, `sub`, `addi`) and memory access (`lw` and `sw`) instructions
- **View register as 32 raw bits**
  - Access individual bits (or groups of bits) rather than the whole 32-bit number
  - Bitwise operations
    - Logical and Shift Operations
    - bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.

Logical Operations

- **Basic logical operators:**
  - AND: outputs 1 only if both inputs are 1
  - OR: outputs 1 if at least one input is 1
  - XOR: outputs 1 if only one input is 1
- **Truth Table: standard table listing all possible combinations of inputs and resultant output for each**

A	B	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

- **Why no NOT?**
  - Can XOR with -1
  - **not t0, t2 can be done with xori t0, t2, -1**

Syntax of Logical Instructions

- **Syntax:**

  1 2,3,4

  1. operation name: and, or, xor, andi, ori, xori
  2. Destination register that will receive result
  3. first operand (register)
  4. second operand (register) or immediate (12-bit numerical constant)

     - immediate values are sign extended!

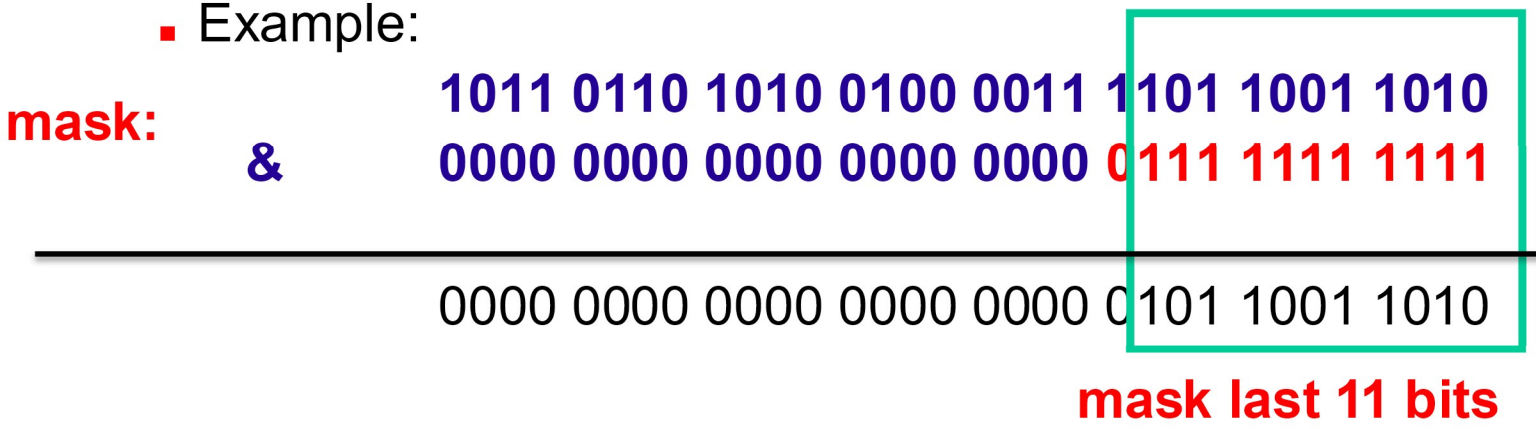
- **RISC-V assembly accepts exactly 2 inputs and produces 1 output**
  - rigid syntax, simpler hardware

Logical Instructions

operation	C	RISC-V
AND	`a = b & c`	`and s0, s1, s2`
OR	`d = e | f`	`or s3, s4, s5`
XOR	`g = h ^ i`	`xor s6, s7, s8`
NOT	`a = ~b`	`xori s0, s1, -1`

Usage of and

- **and-ing a bit with 0 produces a 0 at the output**
- **and-ing a bit with 1 produces the original bit.**
- **can be used to create a mask.**
  - Example:



  - Mask: used to isolate the rightmost 11 bits of the first bitstring by masking out the rest of the string (e.g. setting it to all 0s).

```
andi      t0,t0,0x7FF
```

Usage of or

- **`or`-ing a bit with 1 produces a 1 at the output**
- **`or`-ing a bit with 0 produces the original bit.**
- **or can be used to force certain bits of a string to 1s.**
  - Example: set the rightmost 11 bits to 1, other bits are untouched

1011 0110 1010 0100 0011 1101 1001 1010
| 0000 0000 0000 0000 0000 0111 1111 1111

1011 0110 1010 0100 0011 1111 1111 1111

```
ori t0,t0,0x7FF
```

Exercise

NOR: NOT OR

nor (t0,t1) = not (or (t0,t1))

How would you perform this in RISC-V?

A	B	A nor B
0	0	1
0	1	0
1	0	0
1	1	0

Answer:

```
or  t0, t0, t1
xori t0, t0, -1
```

Immediate Ranges

- **For addi/andi/ori/**
- **Immediate can be at most 12-bit signed integer (i.e., 3 hex digits)**
  - [-2^11, 2^11 -1] = [0xFFFFF800, 0x000007FF] = [-2048, 2047]
- **RARS is weird**
  - Every value represented in hex is interpreted as an unsigned value
    - 0xFFF will be interpreted as 0x00000FFF
    - 0xFFFFFFFF or -1 acceptable, but not 0xFFF
    - allows values in the range [-2048 … 2047], but does not permit hexadecimal values with MSB set to 1 (0x8_ _ … 0xF_ _): "operand is out of range"
  - Bottom line: Use the value that you actually want rather than a truncated one.

Syntax of Shift Instructions

- **Syntax: 1 2,3,4**
  - 1) operation name
  - 2) register that will receive value
  - 3) first operand (register)
  - 4) shift amount (register or constant < 32)
- **RISC-V shift instructions:**
  - sll, slli (shift left logical): shifts left and fills emptied bits with 0s
  - srl, srli (shift right logical): shifts right and fills emptied bits with 0s
  - sra, srai (shift right arithmetic): shifts right and fills emptied bits by sign extending

Example: srli and slli

- **Move (shift) all the bits in a word to the left or right by a number of bits.**
  - srli s0, s0, 8 (shift right by 8 bits)

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

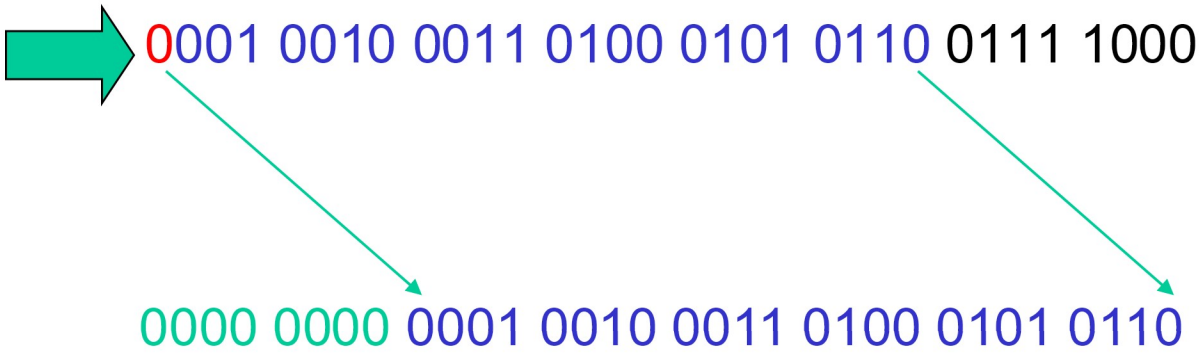
  - slli s0, s0, 8 (shift left by 8 bits)

0001 0010 0011 0100 0101 0110 0111 1000

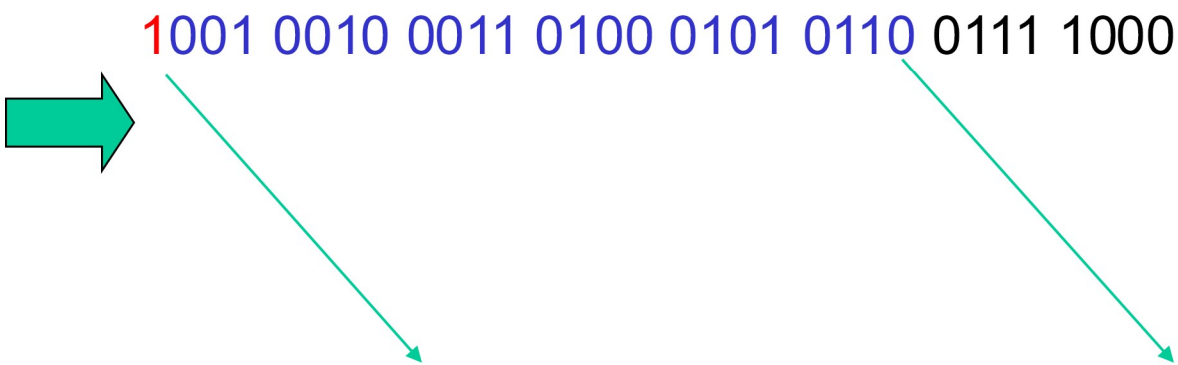
0011 0100 0101 0110 0111 1000 0000 0000

Example: srai

- **srai s0, s0, 8 : shift right arith by 8 bits**



- **srai s0, s0, 8 : shift right arith by 8 bits**



Usage of Shift Instructions

- **shifting left to multiply by powers of 2**
  - a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

    `a *= 8;` (in C)

    would compile to:

    `slli  s0,s0,3` (in RISC-V)
- **shift right to divide by powers of 2**
  - srl/srli with positive/unsigned values only
  - sra/srai rounds down

Shift Left and Overflow

- **For unsigned representation, when the first "1" is shifted out of the left edge, the operation has overflowed.**
  - i.e., the result of the multiplication is larger than the largest possible.

- **Shifting left on signed values also works, but overflow occurs when the most significant bit changes values (from 0 to 1, or 1 to 0).**

How to Set Bits in Upper Half?

- **Example: mask desired is 0xCAFE0000**
- **Solution 1**
  - Build the value in a register: use addi, slli, ori
  - and/or with the register

```
addi t0, zero, 0xCA    # t0 = 0xCA
slli t0, t0, 8         # t0 = 0xCA00
ori  t0, t0, 0xFE      # t0 = 0xCAFE
slli t0, t0, 16        # t0 = 0xCAFE0000
```

- **Solution 2: use lui (load upper immediate) for up to 20 bits**

```
add t0, zero, zero  # t0 = 0x00000000
lui t0, 0xCAFE0      # t0 = 0xCAFE0000
```

Immediate in lui

- **lui s1, 0xFFFAABBD**
  - Assembler: “operation completed with errors”
- **lui s1, 0x123AABBD**
  - Assembler: “operation completed with errors”
- **lui s1, 0xAABB**
  - s1=0xAABB0 000
  - If the immediate is less than 20 bits, 0 will be added on the right
- **Bottom line:**
  - can only take no more than 20 bits of real value in immediate
  - does not sign or zero extend. It works with the immediate directly

How to Set Large Values?

- **Set s1 to hold the value 0xAABBCCDD**

Solution 1:

```
# load and shift each byte
    addi s1, zero, 0x000000AA
    slli s1, s1, 8
    ori s1, s1, 0x000000BB
    slli s1, s1, 8
    ori s1, s1, 0x000000CC
    slli s1, s1, 8
    ori s1, s1, 0x000000DD
```

Solution 2:

```
lui s1, 0xAABBC
lui t0, 0xCDD00
srli t0, t0, 20
or s1, s1, t0
```

Solution 3:

```
lui s1, 0x000AABBD
#Add 1 to 0xAABBD because addi (next instruction) sign extends #(effectively subtracts 1 from the upper value)
addi s1, s1, 0xFFFFFCDD
#FFFFF is to keep the value in range (2^11-1 to -2^11 for two's complement)
```