Constants

- Constants are used frequently in programs
  - In general programming, they represent numerical values that do not change during execution
  - At machine code level, a constant is the use of a numeric value in an instruction in place of a register
- It is not practical to force constants to be in registers before use
  - If loading from memory, only choice is register
  - Lowers performance (takes a memory access)
- RISC-V instruction set has instructions that allow a constant to be specified in the instruction
  - Replaces the second source register operand

Constants

- Size of most used constants are small
  - Example: loop counting, incrementing, decrementing, array indices
- ***Values of constants specified in arithmetic and logic instructions are limited to 12 bits interpreted as 2's complement***
  - ***Numeric range is -2048 to +2047***
- Only one constant can be specified in an instruction
  - The other operands must be registers

opcode dest_reg, src_reg, constant

Immediate Instructions for Constants

- There are *immediate* forms of many instructions
  - Formed by adding "i" to mnemonic
- addi – add immediate
  - Ex. addi t0, t1, 10
    - Which means add the value 10 to the value in register t1 and store the result in register t0
- Not all arithmetic or logic instructions have an immediate format
  - There is ***no subtract immediate*** instruction!
    - Use add immediate and specify negative constant
    - Ex. addi t0, t1, -1 # t0 = t1 – 1
  - Multiply & division also do not have immediate formats

Immediate Instructions (continued)

- Since constants are frequently used, it is beneficial with regard to performance to have immediate instructions for constants
- Design Principle #3: *Always make the common case faster*
- What if you need to specify a constant larger than 12 bits?
  - Ask is it used frequently in the program and what is its scope?
  - Define in memory, load into register once & reuse value being careful not to overwrite the register during program execution
  - Maybe you can create the value in a register using instructions rather than through memory access

Logical Operations

- Logical operations are instructions that allow you to operate on fields of bits within a register or individual bits – called *bitwise* operations
- Original uses for logic operations were to perform packing and unpacking of bit values into words, among other things
- Logical operations are included in many HLLs

Logical Operation	Java Operators
Unsigned shift left	<<
Unsigned shift right	>>>
Signed shift right	>>
Bit-by-bit AND	&
Bit-by-bit OR	|
Bit-by-bit XOR	^
Bit-by-bit NOT	~

Bitwise Logic

- Operations are performed bit by bit
- Logical operations typically defined by truth tables

A	B	&
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	|
0	0	0
0	1	1
1	0	1
1	1	1

OR

A	B	^
0	0	0
0	1	1
1	0	1
1	1	0

XOR

A	~
0	1
1	0

NOT

Logical Instructions

- and logical operation
  - Places a 1 in the result only if both bits of the input operands are 1
  - Used as a *mask* to extract or conceal bits
- or logical operation
  - Places a 1 in the result if either input bit is a 1
  - Used to insert or combine bits into values

```
and    t0, t1, t2

t1 = 0 1 1 0 1 0 1 1
t2 = 0 0 0 0 1 1 1 1
-------------------
t0 = 0 0 0 0 1 0 1 1

or     t0, t1, t2

t1 = 0 1 1 0 0 0 0 0
t2 = 0 0 0 0 1 0 1 1
-------------------
t0 = 0 1 1 0 1 0 1 1
```

Logical Instructions (cont)

- Immediate versions of both allow constant defined in the instruction for one operand
  - The exit instruction uses ori with the zero register and constant 10 to define system call code in register a7
- The constants can be stated in decimal or hex; hex is often preferred

```
andi    t0, t1, 0xf

t1  = 0 1 1 0 1 0 1 1
15 = 0 0 0 0 1 1 1 1
t0  = 0 0 0 0 1 0 1 1

ori     t0, t1, 0xb

t1  = 0 1 1 0 0 0 0 0
11 = 0 0 0 0 1 0 1 1
t0  = 0 1 1 0 1 0 1 1
```

Logical Operations (continued)

- Exclusive Or: xor t0, t1, t2

t1 =
10101111
t2 =
11001100
t0 =
01100011

- Exclusive Or immediate: xori t0, t1, 0xcc

t1 =
10101111
0xcc =
11001100
t0 =
01100011

- Exclusive Or is also called the logical sum
  - a 1 is generated if one of the input bits are 1 but not both
  - a 0 is generated if both inputs are zero or both are 1

Logical Operations (continued)

- RISC-V does not include a NOT instruction!
  - Complementing the bits of a value can be accomplished by using the exclusive-or instruction
- xor the value to be complemented with the value -1
- Ex. xor t0, t1, t2
  - t1 is the value to be complemented
  - t2 contains the value -1
- Ex. xori t0, t1, -1
  - Use the immediate format and specify -1 as a constant

```
t1 =
00001010
t2 =
11111111
t0 =
11110101
```

Shift Operations

- Allows you to move bits in a register left or right
  - A zero bit is shifted into the emptied bit positions on left or right
- Instructions
  - Shift left logical – shift bits left – sll
    - Shift left *i* bits multiplies value by 2^i
    - Shifting left 1 bit multiplies the value by 2
  - Shift right logical – shift bits right – srl
    - Shift right of *i* bits divides value by 2^i
    - Shifting right 1 bit divides value by 2
    - Divide by logical shift right only works for positive values

Shift Operations

- Use of shift instructions is convenient for calculating power of 2 multiplication and division
  - Better performance by avoiding use of more complex hardware in ALU for multiplication and division
- Examples:

20_{10} = 00010100

Shift left 2 bits positions: 01010000 = 80_{10}

20_{10} = 00010100

Shift right 2 bits positions: 00000101 = 5_{10}

Orange bits are the ones shifted out; yellow bits are the ones shifted in.

RISC-V Logical Shift Instructions

- Designed to support both 32 and 64 bit values
- Shift left logical instructions syntax
  - **sll** dest_reg, src_reg, shift_reg
    - shift src_reg left by the value of lower **6** bits in shift_reg (64-bit shift)
  - sllw dest_reg, src_reg, shift_reg
    - shift src_reg left by the value of lower **5** bits in shift_reg (32-bit shift)
  - **slli** dest_reg, src_reg, shift_amt
    - shift src_reg left by the value of constant shift_amt (64-bit shift)
  - slliw dest_reg, src_reg, shift_amt
    - shift src_reg left by the value of constant shift_amt (32-bit shift)

Max 64-bit bit shift of 6 bits; max 32-bit shift of 5 bits.

RISC-V Logical Shift Instructions

- Shift right logical instructions syntax
  - **srl** dest_reg, src_reg, shift_reg
    - shift src_reg right by the value of lower 6 bits in shift_reg (64-bit shift)
  - srlw dest_reg, src_reg, shift_reg
    - shift src_reg right by the value of lower 5 bits in shift_reg (32-bit shift)
  - **srli** dest_reg, src_reg, shift_amt
    - shift src_reg right by the value of constant shift_amt (64-bit shift)
  - srliw dest_reg, src_reg, shift_amt
    - shift src_reg right by the value of lower 5 bits in shift_amt (32-bit shift)

Max 64-bit bit shift of 6 bits; max 32-bit shift of 5 bits.

Shift Right Arithmetic

- As previously noted, shift right logical does not produce the appropriate divide by 2 result for negative numbers
  - Due to the zero taking place of shifted sign bit
- Shift right arithmetic is another shift instruction specifically applied to two's complement shifts where the sign bit needs to be preserved
- The bit shifted in is the same value that previously occupied the most significant bit position
  - If MSB was 0, a 0 is shifted in
  - If MSB was 1, a 1 is shifted in

RISC-V Shift Right Arithmetic Instructions

- Shift right logical instructions syntax
  - **sra** dest_reg, src_reg, shift_reg
    - shift src_reg right by the value of lower 6 bits in shift_reg (64-bit shift)
  - sraw dest_reg, src_reg, shift_reg
    - shift src_reg right by the value of lower 5 bits in shift_reg (32-bit shift)
  - **srai** dest_reg, src_reg, shift_amt
    - shift src_reg right by the value of constant shift_amt (64-bit shift)
  - sraiw dest_reg, src_reg, shift_amt
    - shift src_reg right by the value of lower 5 bits in shift_amt (32-bit shift)

Max 64-bit bit shift of 6 bits; max 32-bit shift of 5 bits.

Shift Right Arithmetic

- Example shifting a negative integer
  - Value -12_{10} = 0xfffffff4 as 32-bit word
    - $11111111111111111111111111110100_2$
  - Value after 1-bit arithmetic shift: -6_{10} = 0xfffffffa
    - $11111111111111111111111111111010_2$
    - Divide by power of 2 now works
- Example shifting a positive integer
  - Value 12_{10} = 0x0000000c as 32-bit word
    - $00000000000000000000000000001100_2$
  - Value after 1-bit arithmetic shift: 6_{10} = 0x00000006
    - $00000000000000000000000000000110_2$
    - Same result as logical shift

Not Enough Registers

- Programs typically have more variables than processors have registers
- The compiler determines what variables need to be in registers during the various portions of a program and generates the data transfer instructions to transfer data at specific times
- The process of transferring variables not needed right away back to memory is called *spilling*
  - "registers spill over into memory"
  - May need to reload them later if needed
  - Frees registers for use with other data items
  - The stack or allocated temporary space is used

Why Keep Data In Registers?

- *Speed* - memory is slower than registers
- The processor can work with data faster if it is in registers
  - Registers are inside the processor and are small
  - Memory is outside the processor and is large
  - The processor can work with two registers at a time
  - Only one operand can be transferred to or from memory at a time
  - The path between memory and the processor is slow (compared to the speed of the processor)
- To achieve highest performance, registers must be used efficiently by compilers and assembly language programmers