Logical / Shift Instructions

Programming

- **High-level languages:**
  - e.g., C, Java, Python
  - Written at higher level of abstraction
- **High-level constructs:** loops, conditional statements, arrays, function calls
- **First, introduce instructions that support these:**
  - Logical operations
  - Shift instructions
  - Multiplication & division
  - Branches & Jumps

Ada Lovelace, 1815-1852

- Wrote the first computer program
- Her program calculated the Bernoulli numbers on Charles Babbage's Analytical Engine
- She was the daughter of the poet Lord Byron

Logical Instructions

- **`and`, `or`, `xor`**
  - `and`: useful for **masking** bits
    - Masking all but the least significant byte of a value:

      0xF234012F AND 0x000000FF = 0x0000002F
  - `or`: useful for **combining** bit fields
    - Combine 0xF2340000 with 0x000012BC:

      0xF2340000 OR 0x000012BC = 0xF23412BC
  - `xor`: useful for **inverting** bits:
    - A XOR -1 = NOT A (remember that -1 = 0xFFFFFFFF)

Logical Instructions: Example 1

Source Registers

s1	0100 0110	1010 0001	1111 0001	1011 0111
s2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly Code

Result

Assembly Code		Result			
`and s3, s1, s2`	s3	0100 0110	1010 0001	0000 0000	0000 0000
`or  s4, s1, s2`	s4	1111 1111	1111 1111	1111 0001	1011 0111
`xor s5, s1, s2`	s5	1011 1001	0101 1110	1111 0001	1011 0111

Logical Instructions: Example 2

Source Values

t3	0011	1010	0111	0101	0000	1101	0110	1111
imm	1111	1111	1111	1111	1111	1010	0011	0100

sign-extended

Assembly Code

```
andi s5, t3, -1484
ori  s6, t3, -1484
xori s7, t3, -1484
```

Result

s5								
s6								
s7								

-1484 = **0xA34** in 12-bit 2's complement representation.

Shift Instructions

Shift amount is in (lower 5 bits of) a register

- `sll`: shift left logical
  - **Example:** `sll t0, t1, t2 # t0 = t1 << t2`
- `srl`: shift right logical
  - **Example:** `srl t0, t1, t2 # t0 = t1 >> t2`
- `sra`: shift right arithmetic
  - **Example:** `sra t0, t1, t2 # t0 = t1 >>> t2`

Immediate Shift Instructions

Shift amount is an immediate between 0 to 31

- `slli`: shift left logical immediate
  - **Example:** `slli t0, t1, 23 # t0 = t1 << 23`
- `srli`: shift right logical immediate
  - **Example:** `srli t0, t1, 18 # t0 = t1 >> 18`
- `srai`: shift right arithmetic immediate
  - **Example:** `srai t0, t1, 5 # t0 = t1 >>> 5`

Multiplication and Division

Multiplication

32 × 32 multiplication → 64 bit result

```
mul s3, s1, s2
```

s3 = lower 32 bits of result

```
mulh s4, s1, s2
```

s4 = upper 32 bits of result, treats operands as signed

{s4,s3} = s1 x s2

Example: s1 = 0x40000000 = 2^{30}; s2 = 0x80000000 = -2^{31}

s1 x s2 = -2^{61} = 0xE0000000 00000000

s4 = 0xE0000000; s3 = 0x00000000

Division

32-bit division → 32-bit quotient & remainder

```
– div   s3, s1, s2   # s3 = s1/s2
– rem   s4, s1, s2   # s4 = s1%s2
```

Example: s1 = 0x00000011 = 17; s2 = 0x00000003 = 3

s1 / s2 = 5

s1 % s2 = 2

s3 = 0x00000005; s4 = 0x00000002

Branches & Jumps

Branching

- Execute instructions out of sequence
- Types of branches:
  - **Conditional**
    - branch if equal (`beq`)
    - branch if not equal (`bne`)
    - branch if less than (`blt`)
    - branch if greater than or equal (`bge`)
  - **Unconditional**
    - jump (`j`)
    - jump register (`jr`)
    - jump and link (`jal`)
    - jump and link register (`jalr`)

We'll talk about these when discuss function calls

Conditional Branching

RISC-V assembly

```
 addi s0, zero, 4         # s0 = 0 + 4 = 4
 addi s1, zero, 1         # s1 = 0 + 1 = 1
 slli s1, s1, 2           # s1 = 1 << 2 = 4
 beq  s0, s1, target      # branch is taken
 addi s1, s1, 1           # not executed
 sub  s1, s1, s0          # not executed

target:                   # label
 add  s1, s1, s0          # s1 = 4 + 4 = 8
```

Labels indicate instruction location. They can't be reserved words and must be followed by a colon (:)

The Branch Not Taken (bne)

```
# RISC-V assembly
    addi    s0, zero, 4         # s0 = 0 + 4 = 4
    addi    s1, zero, 1         # s1 = 0 + 1 = 1
    slli    s1, s1, 2           # s1 = 1 << 2 = 4
    bne     s0, s1, target      # branch not taken
    addi    s1, s1, 1           # s1 = 4 + 1 = 5
    sub     s1, s1, s0          # s1 = 5 - 4 = 1

target:
    add     s1, s1, s0          # s1 = 1 + 4 = 5
```

Unconditional Branching (j)

```
# RISC-V assembly
    j         target          # jump to target
    srai      s1, s1, 2       # not executed
    addi      s1, s1, 1       # not executed
    sub       s1, s1, s0      # not executed

  target:
    add      s1, s1, s0       # s1 = 1 + 4 = 5
```

Conditional Statements & Loops

Conditional Statements & Loops

- **Conditional Statements**
  - `if` statements
  - `if/else` statements
- **Loops**
  - `while` loops
  - `for` loops

If Statement

C Code

```
if (i == j)
  f = g + h;


f = f - i;
```

RISC-V assembly code

```
# s0 = f, s1 = g, s2 = h
# s3 = i, s4 = j
```

Assembly tests opposite case (`i != j`) of high-level code (`i == j`)

If/Else Statement

C Code

```
if (i == j)
  f = g + h;

else
  f = f - i;
```

RISC-V assembly code

```
# s0 = f, s1 = g, s2 = h
# s3 = i, s4 = j
```

Assembly tests opposite case (`i != j`) of high-level code (`i == j`)

While Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

RISC-V assembly code

```
# s0 = pow, s1 = x
```

Assembly tests opposite case (`pow == 128`) of high-level code (`pow != 128`)

For Loops

```
for (initialization; condition; loop operation)
  statement
```

- `initialization:` executes **before** the loop begins
- `condition:` is tested **at the beginning** of each iteration
- `loop operation:` executes at the **end** of each iteration
- `statement:` executes **each time** the condition is met

For Loops

C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

RISC-V assembly code

```
# s0 = i, s1 = sum
```

Less Than Comparison

C Code

```c
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

RISC-V assembly code

```
# s0 = i, s1 = sum
```

Less Than Comparison: Version 2

C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

RISC-V assembly code

```
# s0 = i, s1 = sum
       addi  s1, zero, 0
       addi  s0, zero, 1
       addi  t0, zero, 101
loop:
       slt   t2, s0, t0
       beq   t2, zero, done
       add   s1, s1, s0
       slli  s0, s0, 1
       j     loop
done:
```

slt: set if less than instruction

`slt t2, s0, t0` # if `s0` < `t0`, `t2` = 1

otherwise `t2` = 0

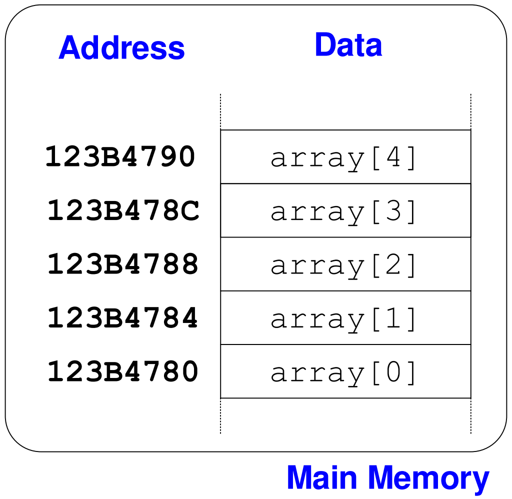
Arrays

Arrays

- Access large amounts of similar data
- **Index**: access each element
- **Size**: number of elements

Arrays

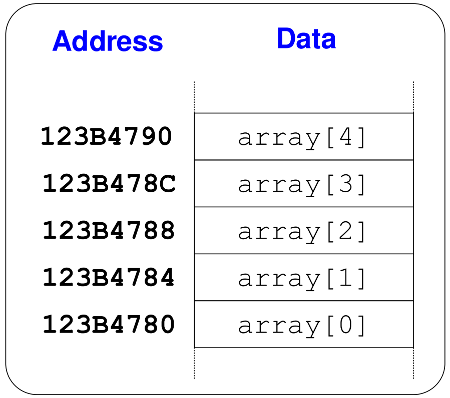
- 5-element array
- **Base address** = 0x123B4780 (address of first element, `array[0]`)
- First step in accessing an array: load base address into a register



Accessing Arrays

```
// C Code
   int array[5];
   array[0] = array[0] * 2;
   array[1] = array[1] * 2;

# RISC-V assembly code
# s0 = array base address
```



Main Memory

Accessing Arrays Using For Loops

```
// C Code
   int array[1000];
   int i;

   for (i=0; i < 1000; i = i + 1)
        array[i] = array[i] * 8;

# RISC-V assembly code
# s0 = array base address, s1 = i
```

Accessing Arrays Using For Loops

```
# RISC-V assembly code
# s0 = array base address, s1 = i
# initialization code
  lui  s0, 0x23B8F          # s0 = 0x23B8F000
  ori  s0, s0, 0x400        # s0 = 0x23B8F400
  addi s1, zero, 0          # i = 0
  addi t2, zero, 1000       # t2 = 1000

loop:
  bge  s1, t2, done         # if not then done
  slli t0, s1, 2            # t0 = i * 4 (byte offset)
  add  t0, t0, s0           # address of array[i]
  lw   t1, 0(t0)            # t1 = array[i]
  slli t1, t1, 3            # t1 = array[i] * 8
  sw   t1, 0(t0)            # array[i] = array[i] * 8
  addi s1, s1, 1            # i = i + 1
  j    loop                 # repeat
done:
```

ASCII Code

- **ASCII:** *American Standard Code for Information Interchange*
- Each text character has unique byte value
  - For example, S = 0x53, a = 0x61, A = 0x41
  - Lower-case and upper-case differ by 0x20 (32)

Cast of Characters: ASCII Encodings

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	**30**	0	**40**	@	**50**	P	**60**	`	**70**	p
21	!	**31**	1	**41**	A	**51**	Q	**61**	a	**71**	q
22	“	**32**	2	**42**	B	**52**	R	**62**	b	**72**	r
23	#	**33**	3	**43**	C	**53**	S	**63**	c	**73**	s
24	$	**34**	4	**44**	D	**54**	T	**64**	d	**74**	t
25	%	**35**	5	**45**	E	**55**	U	**65**	e	**75**	u
26	&	**36**	6	**46**	F	**56**	V	**66**	f	**76**	v
27	‘	**37**	7	**47**	G	**57**	W	**67**	g	**77**	w
28	(	**38**	8	**48**	H	**58**	X	**68**	h	**78**	x
29	)	**39**	9	**49**	I	**59**	Y	**69**	i	**79**	y
2A	*	**3A**	:	**4A**	J	**5A**	Z	**6A**	j	**7A**	z
2B	+	**3B**	;	**4B**	K	**5B**	[	**6B**	k	**7B**	{
2C	,	**3C**	<	**4C**	L	**5C**	\	**6C**	l	**7C**	|
2D	-	**3D**	=	**4D**	M	**5D**	]	**6D**	m	**7D**	}
2E	.	**3E**	>	**4E**	N	**5E**	^	**6E**	n	**7E**	~
2F	/	**3F**	?	**4F**	O	**5F**	_	**6F**	o		

Accessing Arrays of Characters

```
// C Code
   char str[80] = "CAT";
   int len = 0;

   // compute length of string
   while (str[len]) len++;

# RISC-V assembly code
# s0 = array base address, s1 = len
```