



DEPARTMENT OF INFORMATION SYSTEMS AND COMPUTER SCIENCE



```
001011101010001110101110010011101010101001000101
1101010110101010000101010101001010101010101010
101001010010010010101010101010101010101010101010
1110000111101011000000011101010101010000010101
111010101110010100010010111010100010100100111010
10101001010010010010000101010110101010101010010111
0010101001010100101010000001010101001111101000011001
1000110010000111100110101011000100110101010000101010
1100101010101000010011001010100010010101010101010
10100101001001001010101010101010101010101010101010
11100001111010110000000111101010101010000010101
0010010101001010010010100100010101010101001010010
10010100100001010100100101010010100101010010010
1001010010101001010010101001010010101001001001001
10010101010100101010101010010101010101010101010
```

									01
									02
									03
									04
									05
A	B	C	D	E	F	G	H		

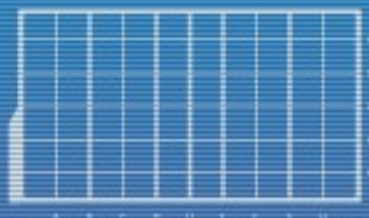
(More) Assembly Programming

Macros, Macros, and More Macros

Lecture Time!

- Macros: Defining Them in BSim
- Arrays: Just Like Ordinary Variables?
- Conversions: Instructions \leftrightarrow 32-bit Lines

```
00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
10010100101010010101010101010101
```

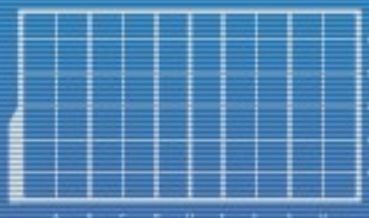


DISCS

Assembly Language

- A symbolic way to write instructions.
- Assembly programs must be translated to *binary machine language* before they can be placed in main memory and executed by the computer.
 - This is done by an assembler.
 - BSim simulates a macro assembler (UASM).

00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
1001010010101001010101010010101



DISCS

Sometimes, it's best to start at the end...

- The Beta CPU is expecting instructions and data in 32-bit (4-byte) chunks.
 - For example, bytes 0-3 form one instruction.
 - Byte 3 is the most significant byte here.
 - The operation code of the instruction can be found in the first 6 bits (which are in byte 3).
 - The address of register C is the next 5 bits (bytes 3 and 2).
 - The address of register A is the next 5 bits (byte 2).
 - Depending on the instruction, the address of register B is the next 5 bits (byte 1) OR the literal is placed in the remaining 16 bits (bytes 1 and 0).
- However, BSim is expecting data to be written per byte.
 - The first byte written is always byte 0.

```
00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
10010100101010010101010101010101
```



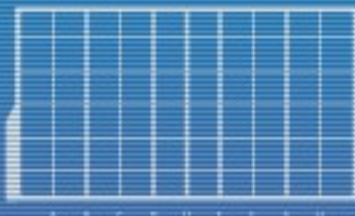
DISCS

Macros

- If you look at beta.uasm, you'll see a LOT of these.
 - It is because of these macros that we can just type
`LD (R31, x, R1)` instead of `0 2 0x3F 0x60`
(assuming x is found in address 0x200).
- First, we need a way to convert one “input” to 4 bytes instead of one input per byte.
 - In other words, we need to define macros! In beta.uasm, you'll find:

```
.macro WORD(x) x%0x100 (x>>8)%0x100  
.macro LONG(x) WORD(x) WORD(x >> 16)
```
- Now, we can type `LONG (2441)` and that 2441 will use four bytes instead of just one!
 - Typing 2441 by itself will yield 137.

00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
1001010010101001010101001010101



DISCS

Macros

- All instructions need to be converted to their 32-bit representations.
 - Luckily, we now have a macro that uses 4 bytes for a given number argument!
- There are two general forms for Beta instructions, so let's start out with those. Again, already in beta.uasm:

```
.macro betaop(OP, RA, RB, RC) {  
    .align 4  
    LONG ( (OP<<26) + ( (RC%0x20) <<21) + ( (RA%0x20) <<16) +  
           ( (RB%0x20) <<11) ) }  
  
.macro betaopc(OP, RA, CC, RC) {  
    .align 4  
    LONG ( (OP<<26) + ( (RC%0x20) <<21) + ( (RA%0x20) <<16) +  
           (CC%0x10000) ) }
```

00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
1001010010101001010101001010101



DISCS

Macros

- Finally, we can define the instructions that will be used.
In beta.uasm:

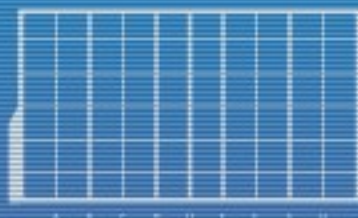
```
.macro ST(RC, CC, RA)    betaopc(0x19, RA, CC, RC)
```

```
.macro BETABR(OP, RA, RC, LABEL)
    betaopc(OP, RA, ((LABEL-.)>>2)-1, RC)
```

```
.macro BNE(RA, LABEL, RC)
    BETABR(0x1E, RA, RC, LABEL)
```

- You can also use macros to contain multiple instruction calls.
 - These macros will be expanded by the assembler, so if a macro contains more than one instruction, you can be sure it will use more than 4 bytes!

00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
10010100101010010101010101010101



DISCS

Arrays

- A contiguous collection of data items where each element is accessed by using an index.
 - Yes, that is correctly spelled.
- The instructions in our programs so far can be considered contiguous, since they are adjacent to each other in memory space.
- A 4-element array named `arr` in BSim:

```
arr: LONG ( 8 )  
LONG ( -3 )  
LONG ( 9999 )  
LONG ( 0 )
```

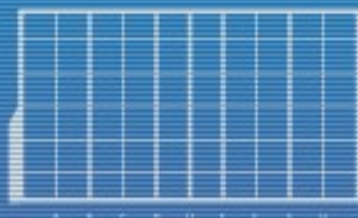
00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
1001010010101001010101001010101



Arrays

- In the previous slide's example, `arr[0] = 8`,
`arr[1] = -3`, and so on.
- Note that even if I label, for example, the 4th element, it
can still be accessed as `arr[3]`.
 - Of course, the question now becomes – How do we
access array elements in Beta?
- First, what does `arr` as a label represent?
- Second, what does the LD instruction's register A
represent?
 - What about both register A and the constant?
- So, what do I do if I want to read `arr[1]`? `arr[2]`?
- What about `arr[x]`, where `x` is a variable?

00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
1001010010100101010101001010101



DISCS

Exercise

- Translate the following code to Beta assembly:

```
int i = 0; // local, just use R1
```

```
while ( i < 4 )
```

```
{ // see previous slides for arr
```

```
    int j = arr[i]; // j is local, use R0
```

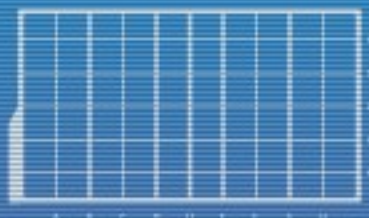
```
    j += i;
```

```
    arr[i] = j;
```

```
    i++;
```

```
}
```

00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
1001010010101001010101001010101

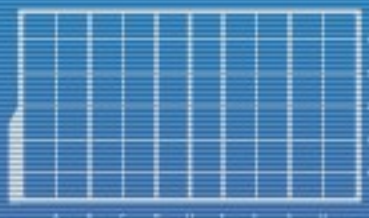


DISCS

Exercises

- Write programs that, given an array of N elements:
 - Replaces each element with the absolute value of that element.
 - Finds the minimum and maximum and puts it in R0 and R1 respectively.
 - Finds the sum of all the elements and puts it in R0.
 - You may treat N as a variable and allocate space for it in your program if so desired.

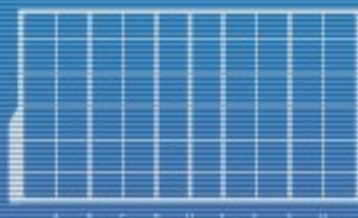
```
00101010010101000011110100001100
10001100100001111001101010010101
11001010101010100001001100101010100
100101001001001010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
10010100101010010101010101010101
```



Lost In Translation

- Part of being able to convert instructions to their 32-bit representations means knowing which bits go where.
 - We already covered this, so I won't repeat it here.
- The most important part of the instruction's 4-byte sequence is its operation code.
 - This OPCODE tells the hardware what exactly it is supposed to do.

00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
1001010010101001010101010010101



DISCS

OPCODE

- In Beta, these 6 bits define the instruction.
 - The other bits define the arguments, if any.
- For example, ADD's opcode is 0b100000, SHRC's opcode is 0b111101, LD's is 0b011000.
- Consult the Beta Instruction Cheat Sheet for a list of operation codes that BSim supports.
 - Most of them must also be supported by your final lab's output.
 - By the way, feel free to print a copy of the cheat sheet --- you'll need it for LT#3.

```
00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
10010100101010010101010101010101
```



Instruction → 32 Bits

- Bits are written starting from the MOST significant.
- Get instruction's OPCODE. (6 bits)
- Get instruction's RC. (5 bits)
- Get instruction's RA. (5 bits)
- Determine type of instruction:
 - Does it require RB? (5 bits)
 - Or does it require a constant? (16 bits)
 - Or neither of the two? (0 bits)
- Set all unused bits to 0 (default Don't Care value).

```
00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
1001010010101001010101010010101
```



DISCS

Instruction → 32 Bits

- SUB (R1, R2, R3)
- SUB's OP CODE = 0b100001
- RC = 3 = 0b00011
- RA = 1 = 0b00001
- RB = 2 = 0b00010
- Result: 0b 100001 00011 00001 00010 000000000000
– 0x 84 61 10 00

00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
10010100101010010101010010101010



DISCS

Instruction → 32 Bits

- LD (R5, 0x200, R0)
- LD's OP CODE = 0b011000
- RC = 0 = 0b000000
- RA = 5 = 0b00101
- Constant = 0x0200 = 0b00000001000000000000
- Result: 0b 011000 00000 00101 000000100000000000
– 0x 60 05 02 00

00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
1110000111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
1001010010101001010101001010101



32 Bits → Instruction

- The process can also be reversed.
- Get first 6 bits. (instruction's OP CODE)
- Get next 5 bits. (instruction's RC)
- Get next 5 bits. (instruction's RA)
- Determine type of instruction:
 - Does it require RB? (5 bits)
 - Or does it require a constant? (16 bits)
 - Or neither of the two? (0 bits)
- Ignore all remaining bits, regardless of value.

```
00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
1001010010101001010101010010101
```

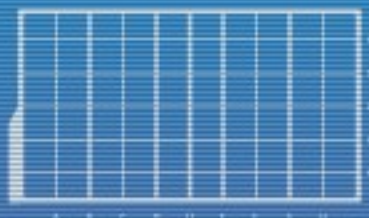


DISCS

32 Bits → Instruction

- 0x BB C1 20 00
- OP CODE = 0b101110 = SRA
- RC = 0b11110 = R30
- RA = 0b00001 = R1
- RB = 0b00100 = R4
- Result: SRA (R1, R4, R30)

00101010010101000011110100001100
10001100100001111001101010010101
110010101010100001001100101010100
1001010010010010101010101010101
11100001111010110000000111101001
001001010100101001001010010010110
10010100100001010100100101001010
10010100101010010100101010010101
1001010010101001010101001010101



DISCS