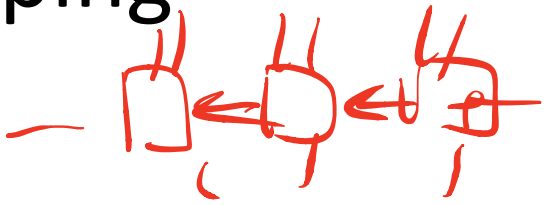
0x0C - MIPS Assembly

ENGR 3410: Computer Architecture

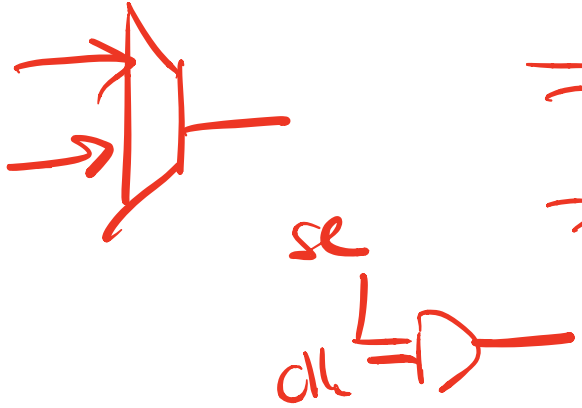
Jon Tse

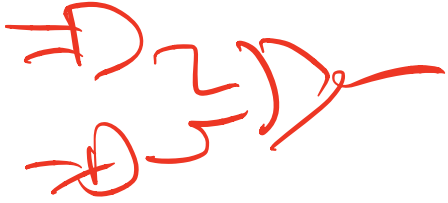
Fall 2020

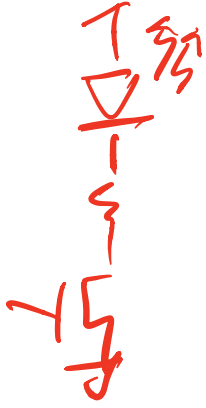
Housekeeping



- Midterm
  - Simplification - Use English, ... , or Verilog
  - Clock Gating - Allowed to use Flops, nothing else
  - LED is black boxed (cut off gfx)
- Lab 4 coming, get MARS running







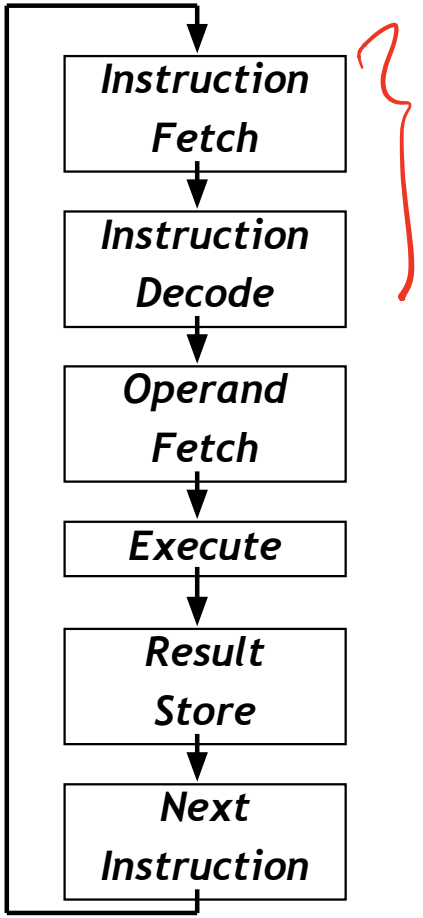
Today

- Review Encodings
- Context of Assembly
- Your First Assembly Programs

What is Assembly?

- Assembly is the lowest level language that humans routinely deal with
- Simple, regular instructions
  - building blocks of C & other languages
- One-to-one mapping to machine language
  - … Usually

What is Assembly?



- It is how we feed steps 1 & 2

Why Shouldn't I learn Assembly?

- Writing in Assembly is usually slower than writing in other languages
  - Time to Market!
- Assembly is not portable
  - Specific to a processor (family)
- Modern Compilers are pretty darn good

Why Should I Learn Assembly?

- Someone has to write those compilers…
- Hand Optimized Assembly can be wicked fast
- Reading Assembly >>> Writing Assembly
- Good practice for designing VMs/protocols
- Initializing an embedded system
  - Setting up Memory, Transitioning execution…

Why Should I Learn Assembly?

- **It will make you a better software architect**

- Optimize from the top down

How should I learn Assembly?

- Generically!
- With a cheat sheet in hand
- Do not memorize specifics

Assembly and Machine Code

- Machine Code can be decompiled to Assembly
- Assembly -> Machine Code -> Assembly loses relatively little information
- C is built on Machine Code

MIPS Assembly Language

- The basic instructions have four components:
  - Operator name
  - Destination
  - 1st operand
  - 2nd operand

```
add <dst>, <src1>, <src2>      # <dst> = <src1> + <src2>
sub <dst>, <src1>, <src2>      # <dst> = <src1> - <src2>
```

- Simple format: easy to implement in hardware

- More complex: `A = B + C + D – E`

Register File Allocation

Register	Name	Function	Comment
$0	$zero	Always 0	No-op on write
$1	$at	Reserved for assembler	Don’t use it!
$2-3	$v0-v1	Function return	
$4-7	$a0-a3	Function call parameters	
$8-15	$t0-t7	Volatile temporaries	Not saved on call
$16-23	$s0-s7	Temporaries (saved across calls)	Saved on call
$24-25	$t8-t9	Volatile temporaries	Not saved on call
$26-27	$k0-k1	Reserved kernel/OS	Don’t use them
$28	$gp	Pointer to global data area	
$29	$sp	Stack pointer	
$30	$fp	Frame pointer	
$31	$ra	Function return address	

Register File Allocation

Register	Name	Function	Comment
$0	$zero	Always 0	No-op on write
$1	$at	Reserved for assembler	Don't use it!
$2-3	$v0-v1	Function return	
$4-7	$a0-a3	Function call parameters	
$8-15	$t0-t7	Volatile temporaries	Not saved on call
$16-23	$s0-s7	Temporaries (saved across calls)	Saved on call
$24-25	$t8-t9	Volatile temporaries	Not saved on call
$26-27	$k0-k1	Reserved kernel/OS	Don't use them
$28	$gp	Pointer to global data area	
$29	$sp	Stack pointer	
$30	$fp	Frame pointer	
$31	$ra	Function return address	

STOP MEMORIZING THIS

Register File Allocation Redux

- 0 is $zero is 0x00000000
  - The only physically special address
  - Specific to MIPS!
- The other 31 are all interchangeable
- But we treat them differently for convenience
- For today, we'll use $t0 through $t9

Basic Operations

Mathematic: add, sub, mul, div `add $t0, $t1, $t2 # t0 = t1+t2`

Unsigned (changes overflow condition) `addu $t0, $t1, $t2 # t0 = t1+t2`

Immediate (one input a constant) `addi $t0, $t1, 100 # t0 = t1+100`

Logical: and, or, nor, xor `and $t0, $t1, $t2 # t0 = t1&t2`

Immediate `andi $t0, $t1, 7 # t0 = t1&b0111`

Shift: left & right logical, arithmetic `sllv $t0, $t1, $t2 # t0 = t1<<t2`

Immediate `sll $t0, $t1, 6 # t0 = t1<<6`

Basic Operations

Mathematic: add, sub, mul, div `add $t0, $t1, $t2  # t0 = t1+t2`
Unsigned (changes overflow condition) `addu $t0, $t1, $t2  # t0 = t1+t2`
Immediate (one input a constant) `addi $t0, $t1, 100  # t0 = t1+100`

Logical: and, or, nor, xor `and $t0, $t1, $t2  # t0 = t1&t2`
Immediate `andi $t0, $t1, 7 # t0 = t1&b0111`

Shift: left & right logical, arithmetic `sllv $t0, $t1, $t2  # t0 = t1<<t2`
Immediate `sll $t0, $t1, 6  # t0 = t1<<6`

Use A Cheat Sheet For These

Example: Jump and Branch

MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Branch On Equal	`beq`	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	`bne`	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	`j`	J	PC=JumpAddr	(5)	2_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

Our First Assembly Program

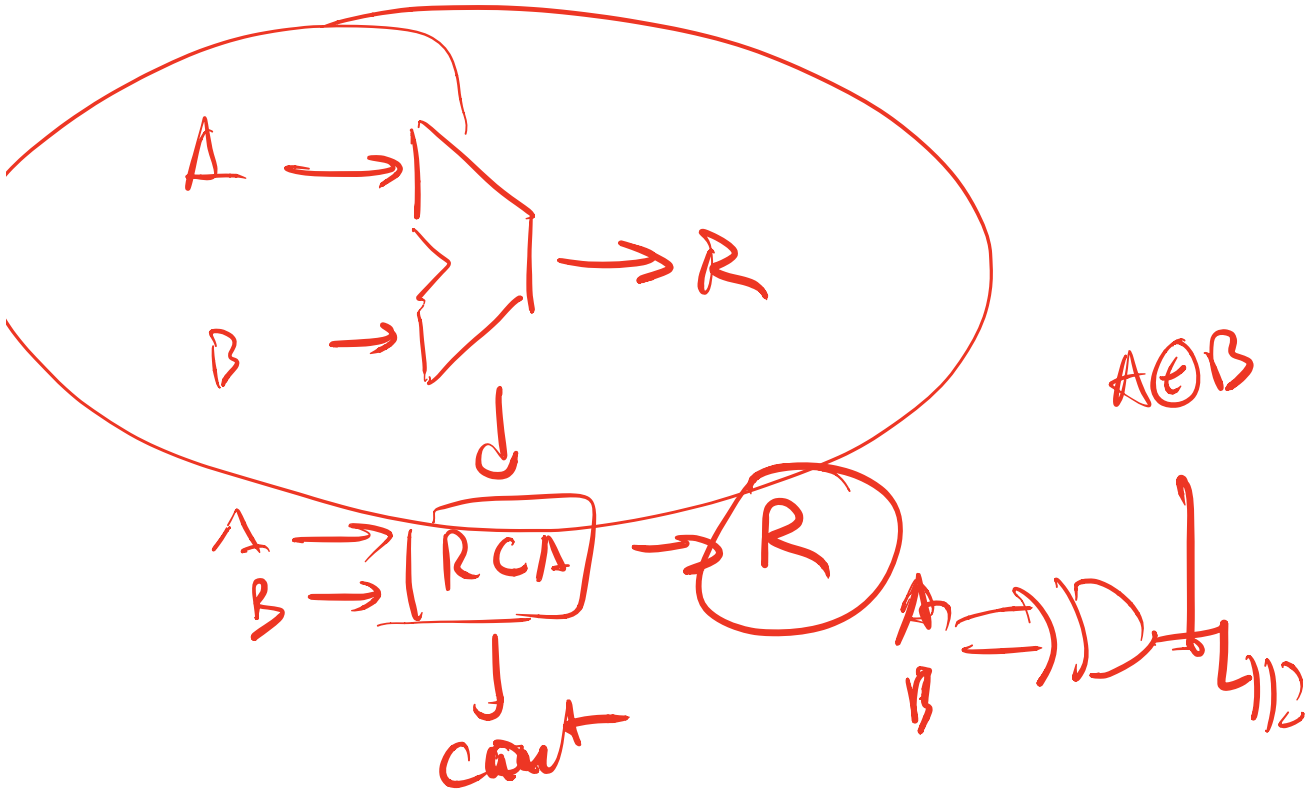
- Average Four Numbers
- $t5 = ($t0 + $t1 + $t2 + $t3)/4
- How do we write this in MIPS Assembly?

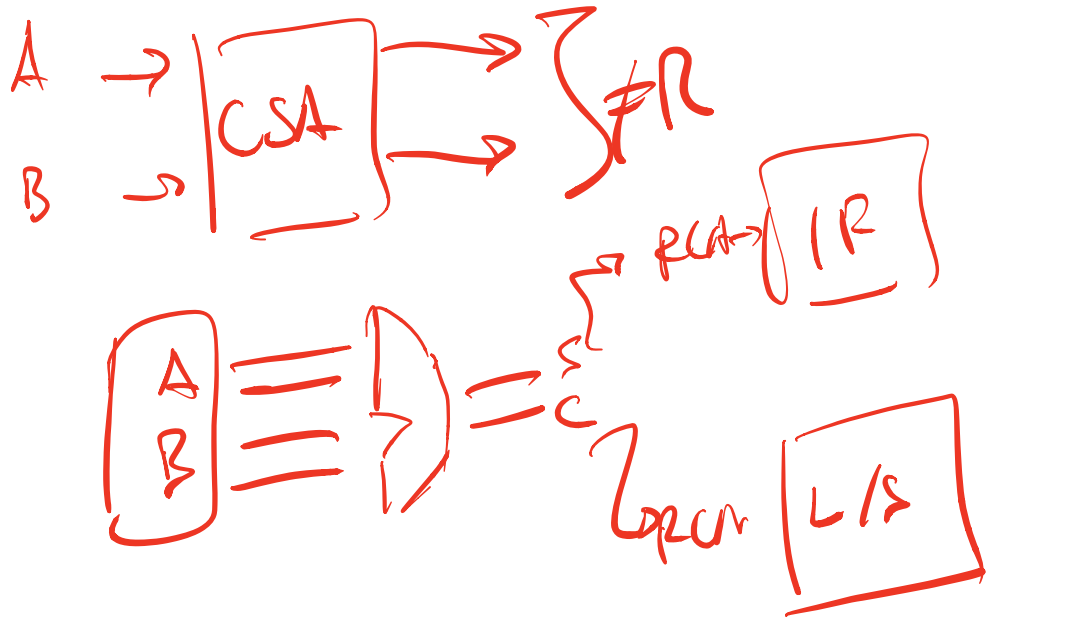
Our First Assembly Program

$t5 = ($t0 + $t1 + $t2 + $t3)/4] not assembly

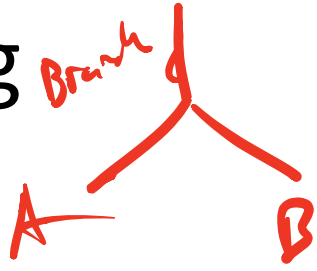
```
add    $t5, $t0, $t1    # $t5 = $t0 + $t1
add    $t5, $t5, $t2    # $t5 += $t2
add    $t5, $t5, $t3    # $t5 += $t3
shift  $t5, $t5, 2      # $t5 /= 4
```

What assumptions are we making for the inputs?
Which shift do we use?


A
B
R
RCA
R
Cout
A⊕B


A
B
CSA
≠R
RCA
1R
A
B
=
Cout
1/2

Jumping and Branching



- Review:
  - Jumps change the Program Counter, period.
  - Branches check a condition and optionally jump.
  - Needs a target location or offset to go to/by

- Calculating jump locations by hand is boring
  - Let the Assembler do it for us!
  - Use Labels

Jumping / Branching Example

```
if (a ==b)
   a+=3
else
   b+=7
c=a+b
```

A simple example to show case jumping and branching

Jumping / Branching Example

```
if (a==b)
   a+=3;
ELSEIF:
   b+=7;
ENDIF:
c=a+b
```

- Add labels to indicate our branch / jump targets

Jumping / Branching Example

```
if (a !=b) goto
ELSEIF
   a+=3;
   goto ENDIF
ELSEIF:
   b+=7;
ENDIF:
c=a+b
```

Rewrite our if statements

If (x) do y becomes

If (~x) skip y

Jumping / Branching Example

```
bne a, b, ELSEIF
   a+=3;
   j ENDIF
ELSEIF:
   b+=7;
ENDIF:
c=a+b
```

- Use specific jump and branch operations

Jumping / Branching Example

```
bne a, b, ELSEIF
  add a, a, 3
   j ENDIF
ELSEIF:
   add b, b, 7
ENDIF:
add c, a, b
```

- Translate remaining code
- Todo:
  - Assign registers to a,b,c
    - Human Job
  - Calculate offsets
    - Assembler Job

MIPS Assembly vs High Level Languages

- All operations on registers, except load/store
- Simple instructions with 1-2 operands
- No scope, nesting, variables, loops, functions
- Explicit flow control via branch & jump
  - Labels represent an address in data or instruction memory

int c = #;

Assembly Simulator

- Download MARS (already on the docker)

  http://courses.missouristate.edu/kenvollmar/mars/

  http://goo.gl/mOsZ8

  sudo apt install default-jre #not on the docker

  java -jar <MARS_JAR_FILE>

- Open the environment, start writing assembly

- Hint: Work through the process methodically

Desk/Board Work

- Calculate the Sum of integers [0,N]
- Calculate Nth Fibonacci Number

- Hints:
  - N starts in $a0, put answer in $v0
  - Use $t0-$t9 as temporary storage
  - Set initial conditions with `addi <dest>, $zero, value`
  - Comment with #
- Challenge:
  - Multiply without using mul, mult, etc
  - Find Square Root (with a loop? binary search?)
  - Find Nth Prime

Fibonacci Solution with Loops

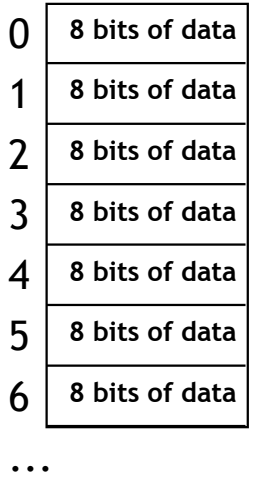
```
addi $a0, $zero, 5      #assume N>=2 argument provided in a0

addi $t0, $zero, 1      #F_(n-1)
addi $t1, $zero, 1      #F_(n-2)
addi $t2, $zero, 2      #n
                #t3 is F_n

loop:
    add $t3,$t0,$t1         #F_n=F_(n-1)+F_(n-2)
    beq $t2,$a0,breakloop   #if (n==N) goto breakloop;
    addi $t2,$t2,1          #n++
    add $t1,$zero,$t0       #F_(n-2)=F_(n-1)
    add $t0,$zero,$t3       #F_(n-1)=F_n
    j loop              #restart loop

breakloop:
    add $v0,$zero,$t3       #return the answer
```

Memory Organization

- Lets talk to Data Memory!

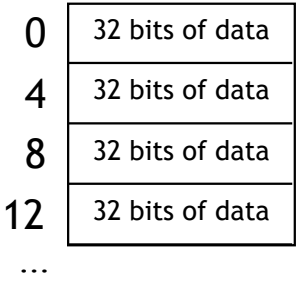
mem[i]

- Viewed as an 1-dimension array
- Memory address indexes into array
- Byte Addressable:
  - 8 bits of data per address



Memory Organization

- Bytes are nice, but our registers are "words"
- For MIPS, a word is 32 bits or 4 bytes.



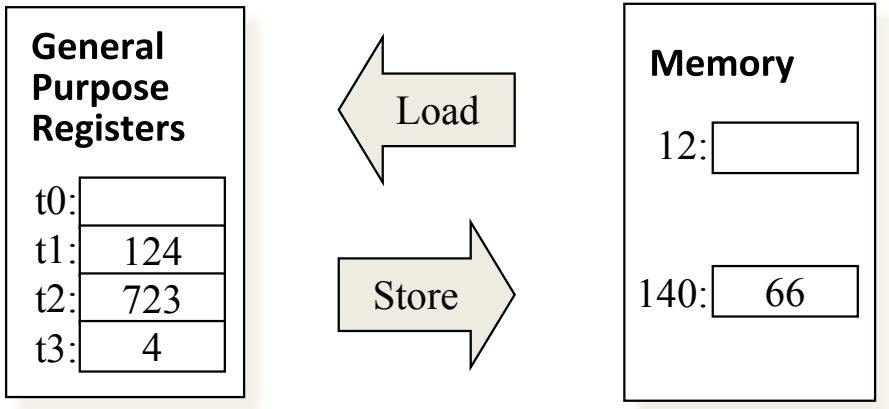
Our registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4

Loads & Stores

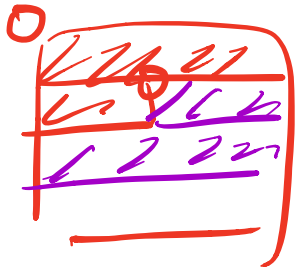
- Loads & Stores move words between memory and registers
  - All operations on registers, but too small to hold all data

```
lw $t0, $t1     # $t0 = Memory[$t1]
sw $t2, $t3     # Memory[$t3] = $t2
```

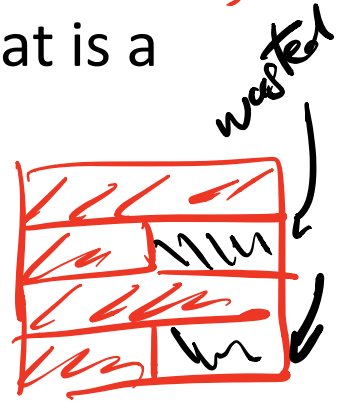


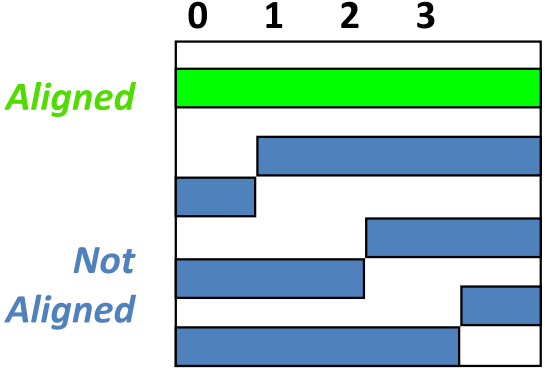
- Note: lbu & sb load & store bytes

Word Alignment



- Require that objects fall on an address that is a multiple of their size

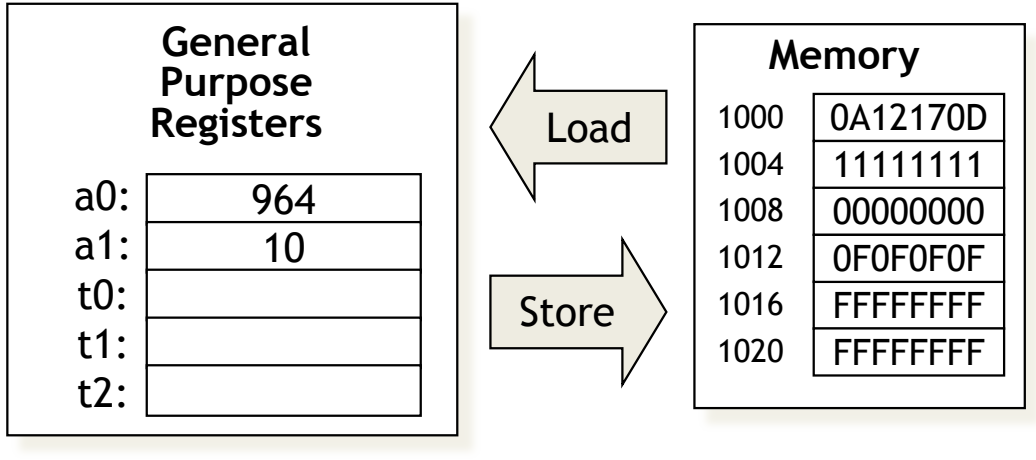




- What are the last two bits of a 32-bit word aligned address?

Array Example

```
/* Swap the kth and (k+1)th
   element of an array */
swap(int v[], int k)
{
    int temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}


# Assume v in $a0,
  k in $a1
```



Array Example

```
/* Swap the kth and (k+1)th
   element of an array */
swap(int v[], int k)
{
    load v[k] into temp0
    load v[k+1] into temp1
    store temp0 into v[k+1]
    store temp1 into v[k]
}
```

```
# Assume v in $a0,
  k in $a1
```

Convert array operations in to load / store operations

Why do we need temp1?

Why wasn't it in the C code?

Array Example

```
/* Swap the kth and (k+1)th
   element of an array */
swap(int v[], int k)
{
    address0 = v+k*4
    address1 = address0 + 4
    load address0 into temp0
    load address1 into temp1
    store temp0 into address1
    store temp1 into address0
}
```

```
# Assume v in $a0,
  k in $a1
```

Calculate memory address of v[k] as

v + k * sizeof(int)

v + k * 4

Array Example

```
/* Swap the kth and (k+1)th
   element of an array */
swap(int v[], int k)
{
    address0 = k*4
    address0+= v
    address1 = address0 + 4
    load address0 into temp0
    load address1 into temp1
    store temp0 into address1
    store temp1 into address0
}


# Assume v in $a0,
  k in $a1
```

Expand math for calculating address0

Array Example

```
/* Swap the kth and (k+1)th
   element of an array */
swap(int v[], int k)
{
    sll address0, k, 2
    add address0, address0, v
    add address1, address0, 4
    lw temp0, (address0)
    lw temp1, (address1)
    sw temp0, (address1)
    sw temp1, (address0)
}
```

```
# Assume v in $a0,
  k in $a1
```

Replace with actual instructions

sll – shift left

lw – load word

sw – store word

Todo:

Assign Registers

Array Example

```
/* Swap the kth and (k+1)th
   element of an array */
swap(int v[], int k)
{
    sll address0, k, 2
    add address0, address0, v
    add address1, address0, 4
    lw temp0, (address0)
    lw temp1, (address1)
    sw temp0, (address1)
    sw temp1, (address0)
}


# Assume v in $a0,
  k in $a1
```

```
/* Swap the kth and (k+1)th
   element of an array */
swap(int v[], int k)
{
    int temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}


# Assume v in $a0,
  k in $a1
```

Array problems

- Practice Assembly
  - Can you reverse an array?
  - Can you sort an array?
  - How do you multiply in IQ?
    - Signed? Unsigned?
  - Calculate a Polynomial?

Hint: Prepopulating Arrays

```
… code code code…
lw $t0, array1($t1)
…snip more code…
.data
array1:
0x12345678      #element0
0x23456789      #element1
…
```

Extra: Fixed Point Math in Assembly

- Total number of bits is usually:
  - Full Word (I+Q=32)
  - Half Word (I+Q=16)
- Add/Sub are very easy
  - Use Integer operations unchanged
- Multiplication is mildly tricky
  - May require additional operations
- Division is a pain

Fixed Point Math in Assembly

- How do you Multiply?
  - Integer Multiplication
  - Shift it back in place
  - Overflow?

- For now, use I8Q8 sign extended
  - Top 16 bits are all zero or all one

Fixed Point Math in Assembly

- Multiply U8Q8 x U8Q8
  - Becomes U16Q16
- Shift by 8
  - Becomes U16Q8 with 16 leading zero bits
- And 0x0000FFFF
  - Becomes U8Q8 with 16 leading zero bits