An Introduction to Computing with a Simple Machine

Adapted, with permission from materials developed by:

Dr. Mike Feely, UBC

The CPU

- Supports a set of simple instructions
  - Each instruction would be implemented using logic gates
  - The fewer and the simpler the better
- For example, our CPU should be able to add
  - We need an instruction that does something like: c=a+b
  - a,b and c are the values (the data) in memory
  - The instructions necessary to perform c=a+b (the program) are encoded as a set of bits in memory
    - What do you think are the necessary part of the instruction?
      - Name of the operation and where a, b and c are
  - the instruction might look something like this (in hex):

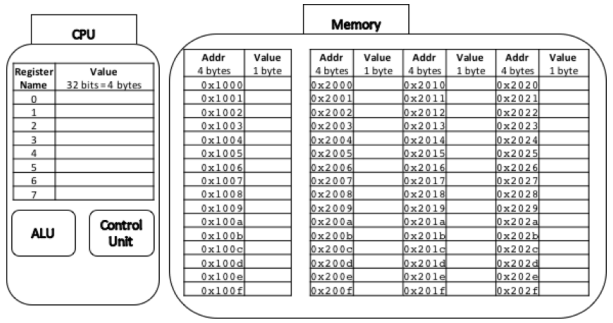
Op code for **add**	Address of a	Address of b	Address of c
01	0x00001000	0x00001004	0x00001008

The need for efficiency

- Accessing memory is slow relative to CPU execution
  - ~1/100 cycles
  - Avoid memory access when possible
- Big instructions are costly
  - Memory addresses are big,
    making instructions that access them big too
  - CPU can cache instructions
    - If enough are cached, CPU will rarely wait for an instruction from memory
  - The smaller the instructions, the more fit in cache
- What would you suggest to mitigate this problem with this large instruction size?

Op code for **add**	Address of a	Address of b	Address of c
01	0x00001000	0x00001004	0x00001008

Recall our architecture:



- Registers hold data the size of an int and the size of an address (32 bits/4 bytes)
- Fast access (roughly single cycle access)
- Have instructions (add, etc), except load and store, operate on data that is already in registers
- We can encode addresses for 8 registers with one hexadecimal digit (4 bits)

	op code for **add**	Address of a	Address of b	Address of c
We can go from this…	01	0x00001000	0x00001004	0x00001008
To this…	01	0x0	0x1	0x2
Or even this (a=a+b)…	01	0x0	0x1	

Instruction Set Architecture (ISA)

- The interface to the processor
  - How to instruct the processor
- Types of instructions
  - Math (add, and, or, not…)
  - Memory access (load, store)
  - Control transfer (gotos)
- Instruction format
  - An opcode + operands (operation + input)
- Assembly language
  - Higher level representation of machine code

RTL – Register Transfer Language

- Transferring
  - memory to registers,
  - register to register,
  - register to memory
- Can't transfer
  - Memory to memory
  - Constant values directly to memory
- RTL == pseudocode
  - not runnable
- Explain in English what you think the following RTL does:
  - r[0] ← 10
  - r[1] ← m[r[0]]
  - r[2] ← r[0] + r[1]

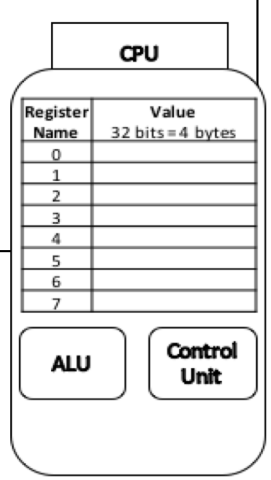
Convert the following program to RTL

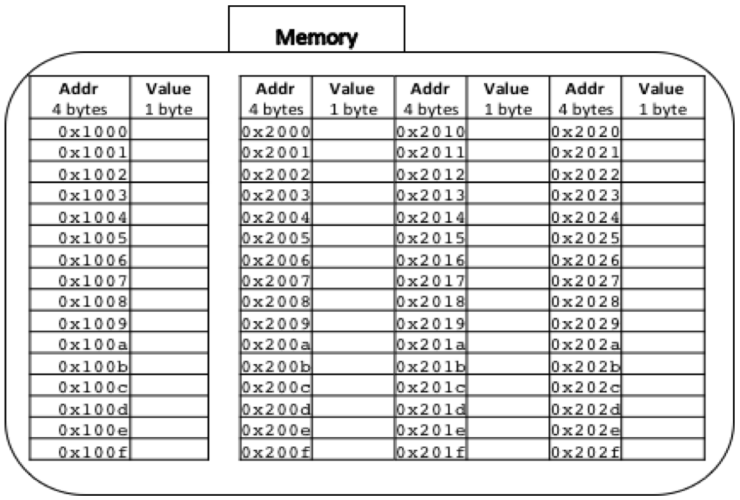
C

```
int a;          // an int
int b[10];  // an array of 10 ints

void foo () {
  a      = 2;
  b[a] = a;
}
```

Assume:
the address of a is 0x1000
the address of b is 0x2000





ASM Language Specification

Name	Semantics	Assembly	Machine
load immediate	`r[d] ← v`	`ld $v, rd`	`0d-- vvvvvvvv`
load base+offset	`r[d] ← m[(o=p*4)+r[s]]`	`ld 0(rs), rd`	`1psd`
load indexed	`r[d] ← m[r[s]+4*r[i]]`	`ld (rs,ri,4), rd`	`2sid`
store base+offset	`m[(o=p*4)+r[d]] ← r[s]`	`st rs, 0(rd)`	`3spd`
store indexed	`m[r[d]+4*r[i]] ← r[s]`	`st rs, (rd,ri,4)`	`4sdi`
register move	`r[d] ← r[s]`	`mov rs, rd`	`60sd`
add	`r[d] ← r[d] + r[s]`	`add rs, rd`	`61sd`
and	`r[d] ← r[d] & r[s]`	`and rs, rd`	`62sd`
inc	`r[d] ← r[d] + 1`	`inc rd`	`63-d`
inc address	`r[d] ← r[d] + 4`	`inca rd`	`64-d`
dec	`r[d] ← r[d] - 1`	`dec rd`	`65-d`
dec address	`r[d] ← r[d] - 4`	`deca rd`	`66-d`
not	`r[d] ← ~ r[d]`	`not rd`	`67-d`
shift left	`r[d] ← r[d] << s`	`shl $s, rd`	`71ss`
shift right	`r[d] ← r[d] >> -s`	`shr $s, rd`	
halt	*halt machine*	`halt`	`F0--`
nop	*do nothing*	`nop`	`FF--`