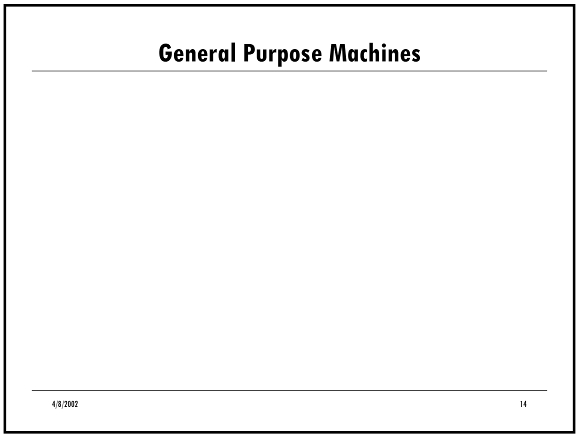
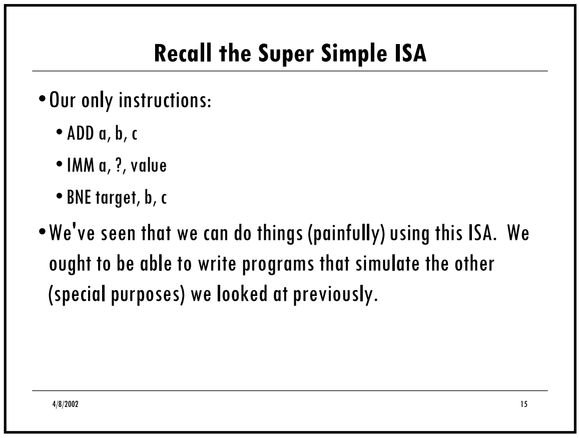
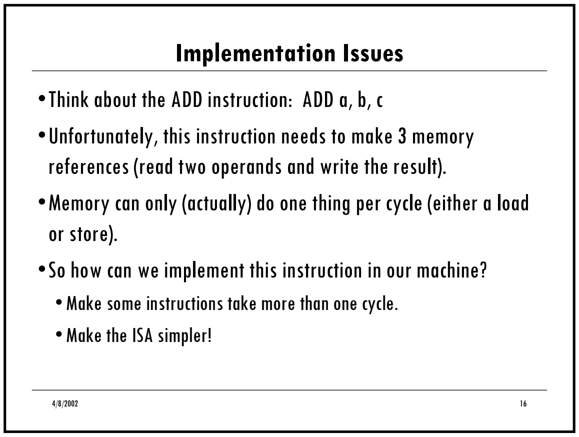


General Purpose Machines



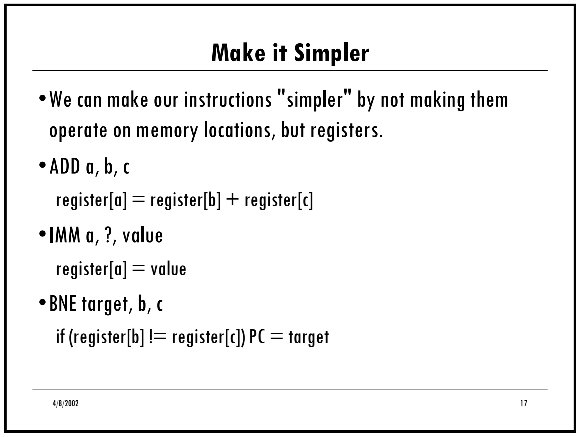
Recall the Super Simple ISA

- Our only instructions:
  - ADD a, b, c
  - IMM a, ?, value
  - BNE target, b, c
- We've seen that we can do things (painfully) using this ISA. We ought to be able to write programs that simulate the other (special purposes) we looked at previously.



Implementation Issues

- Think about the ADD instruction: ADD a, b, c
- Unfortunately, this instruction needs to make 3 memory references (read two operands and write the result).
- Memory can only (actually) do one thing per cycle (either a load or store).
- So how can we implement this instruction in our machine?
  - Make some instructions take more than one cycle.
  - Make the ISA simpler!



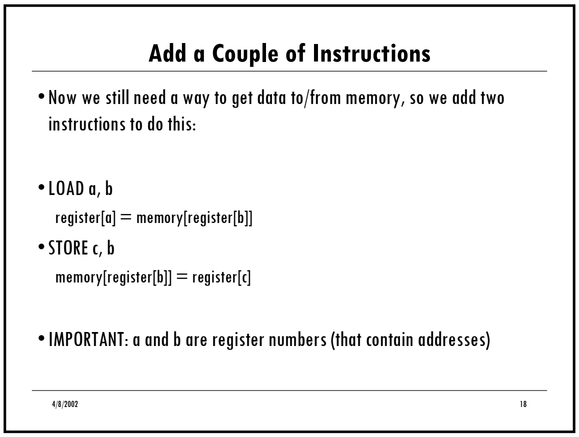
Make it Simpler

- We can make our instructions "simpler" by not making them operate on memory locations, but registers.
- ADD a, b, c

  register[a] = register[b] + register[c]
- IMM a, ?, value

  register[a] = value
- BNE target, b, c

  if (register[b] != register[c]) PC = target

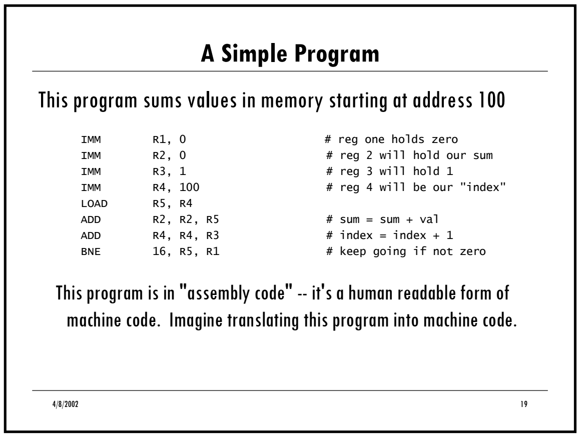


Add a Couple of Instructions

- Now we still need a way to get data to/from memory, so we add two instructions to do this:
- LOAD a, b

  register[a] = memory[register[b]]
- STORE c, b

  memory[register[b]] = register[c]
- IMPORTANT: a and b are register numbers (that contain addresses)

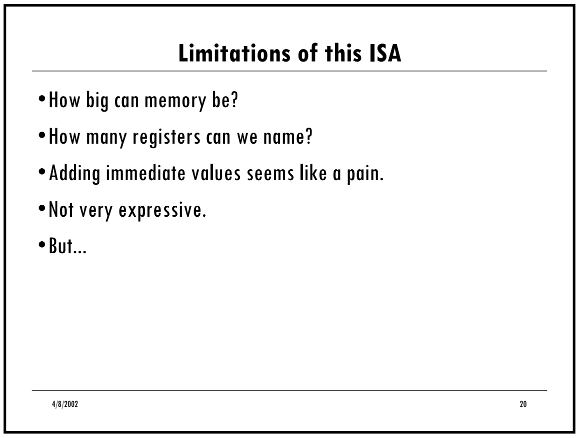


A Simple Program

This program sums values in memory starting at address 100

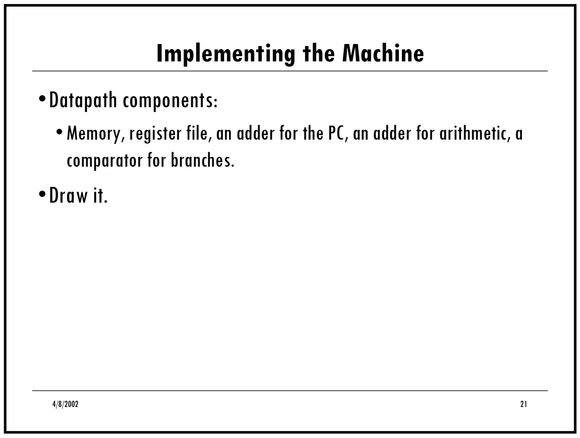
```
IMM     R1, 0              # reg one holds zero
IMM     R2, 0              # reg 2 will hold our sum
IMM     R3, 1              # reg 3 will hold 1
IMM     R4, 100            # reg 4 will be our "index"
LOAD    R5, R4
ADD     R2, R2, R5         # sum = sum + val
ADD     R4, R4, R3         # index = index + 1
BNE     16, R5, R1         # keep going if not zero
```

This program is in "assembly code" -- it's a human readable form of machine code. Imagine translating this program into machine code.



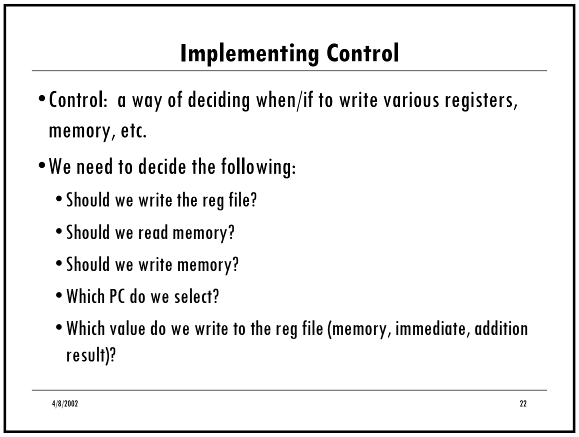
Limitations of this ISA

- How big can memory be?
- How many registers can we name?
- Adding immediate values seems like a pain.
- Not very expressive.
- But…



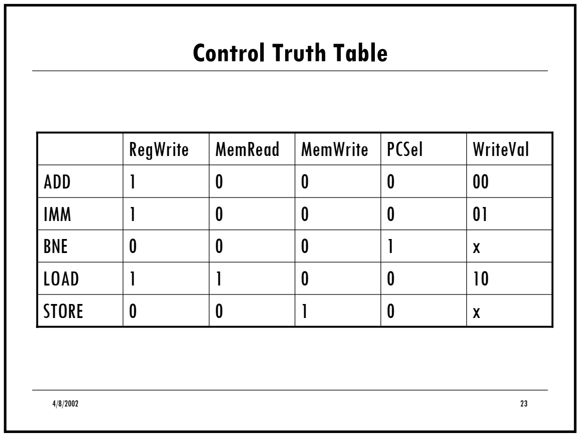
Implementing the Machine

- Datapath components:
  - Memory, register file, an adder for the PC, an adder for arithmetic, a comparator for branches.
- Draw it.



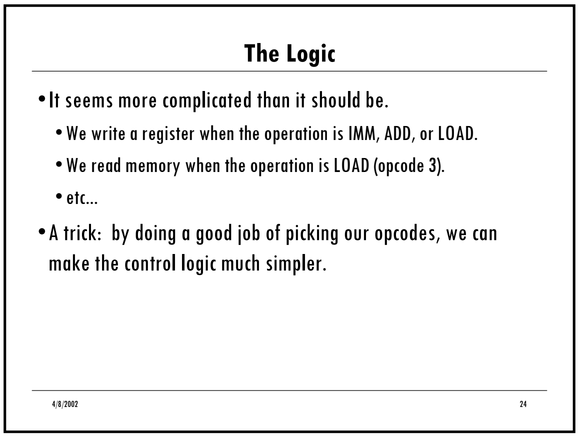
Implementing Control

- Control: a way of deciding when/if to write various registers, memory, etc.
- We need to decide the following:
  - Should we write the reg file?
  - Should we read memory?
  - Should we write memory?
  - Which PC do we select?
  - Which value do we write to the reg file (memory, immediate, addition result)?



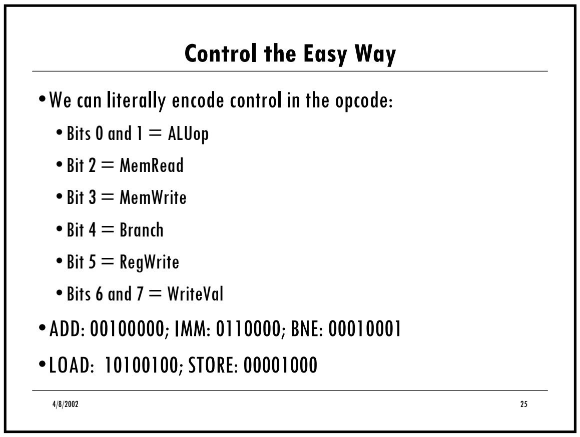
Control Truth Table

	RegWrite	MemRead	MemWrite	PCSel	WriteVal
ADD	1	0	0	0	00
IMM	1	0	0	0	01
BNE	0	0	0	1	x
LOAD	1	1	0	0	10
STORE	0	0	1	0	x



The Logic

- It seems more complicated than it should be.
  - We write a register when the operation is IMM, ADD, or LOAD.
  - We read memory when the operation is LOAD (opcode 3).
  - etc…
- A trick: by doing a good job of picking our opcodes, we can make the control logic much simpler.



Control the Easy Way

- We can literally encode control in the opcode:
  - Bits 0 and 1 = ALUop
  - Bit 2 = MemRead
  - Bit 3 = MemWrite
  - Bit 4 = Branch
  - Bit 5 = RegWrite
  - Bits 6 and 7 = WriteVal
- ADD: 00100000; IMM: 0110000; BNE: 00010001
- LOAD: 10100100; STORE: 00001000