CS184a:
Computer Architecture
(Structure and Organization)

Day 5: January 17, 2003

What is required for Computation?

Last Time

- Generalize compute elements/datapaths for reuse
- Instructions to tell datapaths how to behave
- Memories pack state compactly
- Memories can serve as programmable control
- Virtualize computing operators
  - store descript. and state compactly in memory
  - generalize and reuse datapath

Today

- Review Datapath Operation
- Memory
  - unbounded
  - impact on computability
- Computing Requirements (Review)

Defining Terms

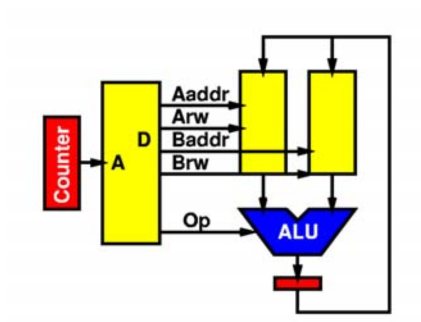
Fixed Function:

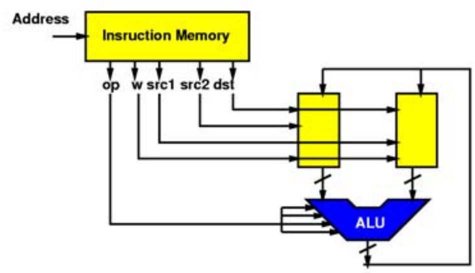
- Computes one function (e.g. FP-multiply, divider, DCT)
- Function defined at fabrication time

Programmable:

- Computes “any” computable function (*e.g.* Processor, DSPs, FPGAs)
- Function defined after fabrication

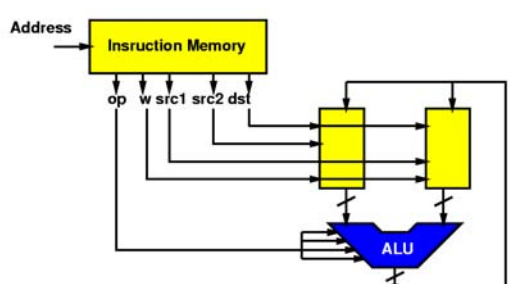
Programmable Memory Control





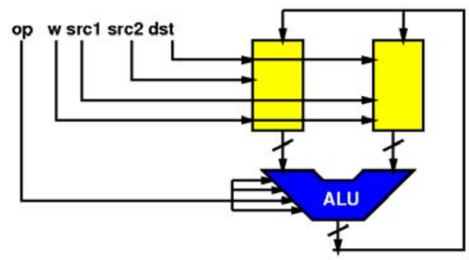
Programmable Memory Control

- Use two memories as cheap dual-ported memory
- Read independently
- Write to both



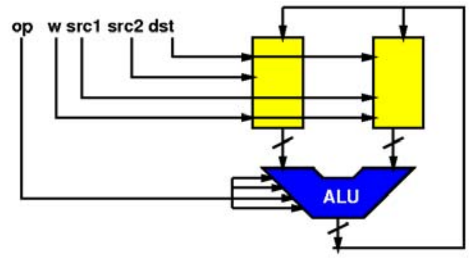
Programming an Operation

- Consider:
  - C = (A+2B) & 00001111
- Cannot do this all at once
  - But can do it in pieces



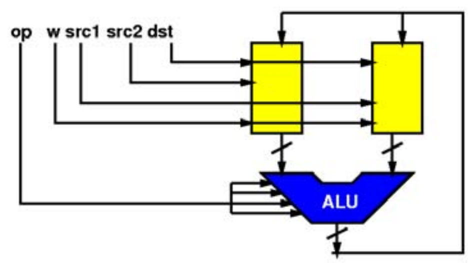
Programming an Operation

- Consider: C = (A+2B) & 00001111
- Find a place for A, B, C
  - A – slot 0
  - B – slot 1
  - C – slot 7
  - 00001111 – slot 4



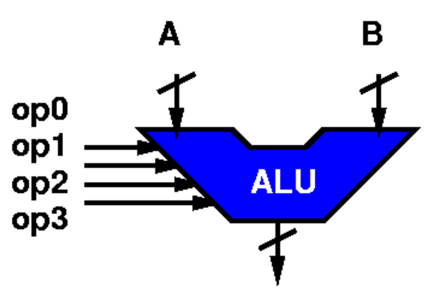
Programming an Operation

- Consider: C = (A+2B) & 00001111
- Decompose into pieces
  - Compute 2B
  - Add A and 2B
  - AND sum with mask



ALU Encoding

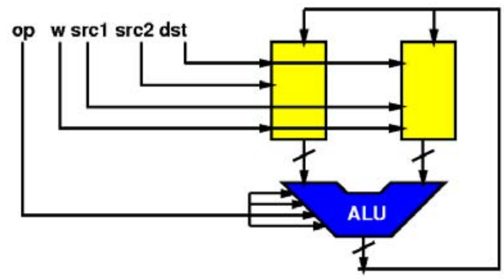
- Each operation has some bit sequence
- ADD 0000
- SUB 0010
- INV 0001
- SLL 1110
- SLR 1100
- AND 1000



Programming an Operation

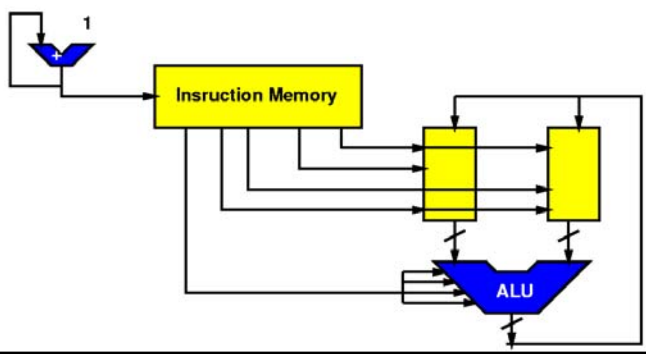
- Decompose into pieces

	Op	w	src1	src2	dst
Compute 2B	0000	1	001	001	010
Add A and 2B	0000	1	000	010	011
AND sum with mask	1000	1	011	100	111



Instruction Control

- Add a counter to sequence through operations



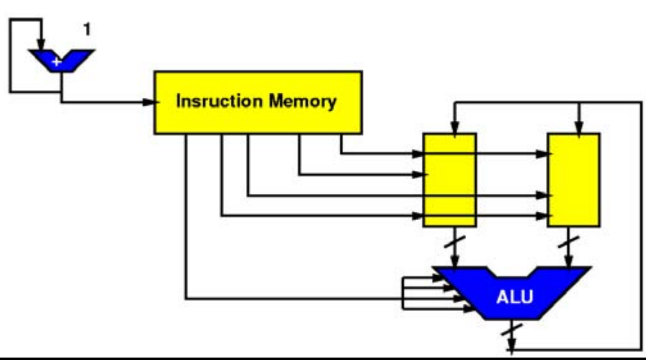
Programming the Operation

- Consider:
  - C = (A+2B) & 00001111
- Decompose into pieces
  - Compute 2B 0000 1 001 001 010
  - Add A and 2B 0000 1 000 010 011
  - AND sum with mask 1000 1 011 100 111
- Now becomes the task of filling in the memory

Instruction Control

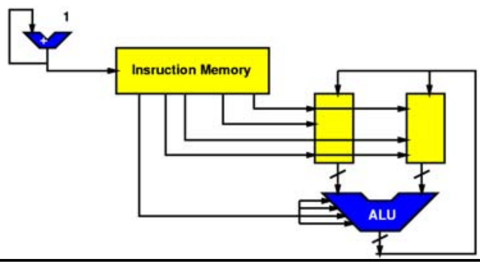
Op w src1 src2 dst

- 000: 0000 1 001 001 010
- 001: 0000 1 000 010 011
- 010: 1000 1 011 100 111



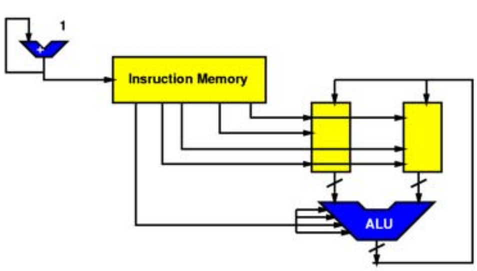
Executing the Program

- To execute program
  - Keep track of state of machine
    1. Value of counter
    2. Contents of instruction memory
    3. Contents of data memory



Machine State: Initial

- Counter: 0
- Instruction Memory:
  - 000: 0000 1 001 001 010
  - 001: 0000 1 000 010 011
  - 010: 1000 1 011 100 111



- Data Memory:
  - 000: A
  - 001: B
  - 010: ?
  - 011: ?
  - 100: 00001111
  - 101: ?
  - 110: ?
  - 111: ?

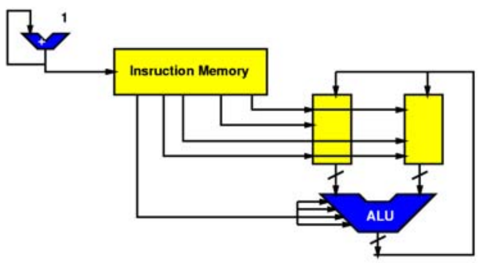
First Operation

- Counter: 0
- Instruction Memory:

  000: 0000 1 001 001 010

  001: 0000 1 000 010 011

  010: 1000 1 011 100 111



- Data Memory:

  000: A

  001: B

  010: ?

  011: ?

  100: 00001111

  101: ?

  110: ?

  111: ?

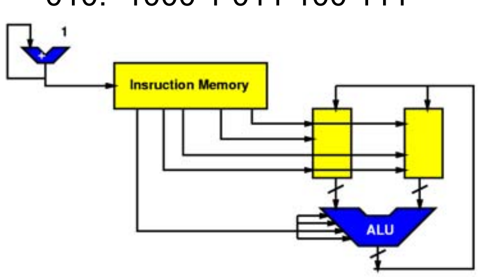
First Operation Complete

- Counter: 0
- Instruction Memory:

  000: 0000 1 001 001 010

  001: 0000 1 000 010 011

  010: 1000 1 011 100 111



- Data Memory:

  000: A

  001: B

  010: **2B**

  011: ?

  100: 00001111

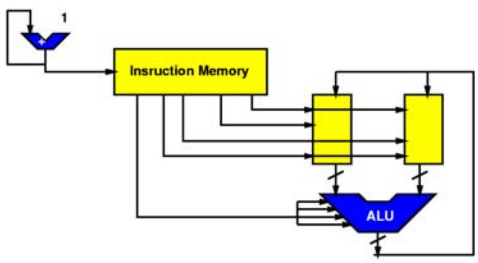
  101: ?

  110: ?

  111: ?

Update Counter

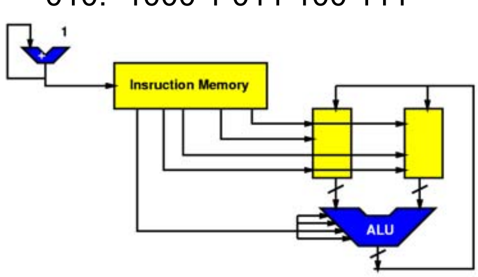
- Counter: 1
- Instruction Memory:
  - 000: 0000 1 001 001 010
  - 001: 0000 1 000 010 011
  - 010: 1000 1 011 100 111



- Data Memory:
  - 000: A
  - 001: B
  - 010: 2B
  - 011: ?
  - 100: 00001111
  - 101: ?
  - 110: ?
  - 111: ?

Second Operation

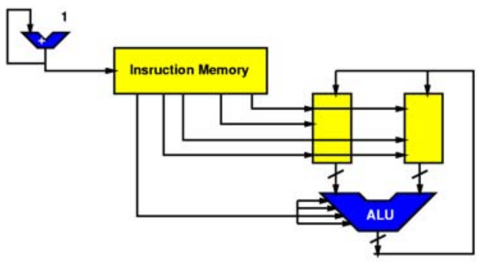
- Counter: 1
- Instruction Memory:
  - 000: 0000 1 001 001 010
  - **001: 0000 1 000 010 011**
  - 010: 1000 1 011 100 111



- Data Memory:
  - 000: A
  - 001: B
  - 010: 2B
  - 011: ?
  - 100: 00001111
  - 101: ?
  - 110: ?
  - 111: ?

Second Operation Complete

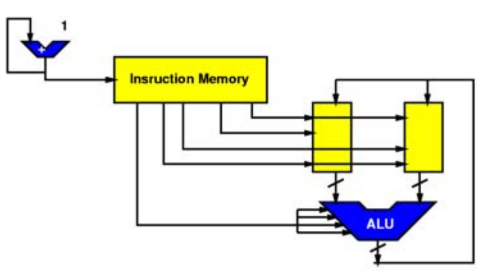
- Counter: 1
- Instruction Memory:
  000: 0000 1 001 001 010
  001: 0000 1 000 010 011
  010: 1000 1 011 100 111



- Data Memory:
  000: A
  001: B
  010: 2B
  011: **A+2B**
  100: 00001111
  101: ?
  110: ?
  111: ?

Update Counter

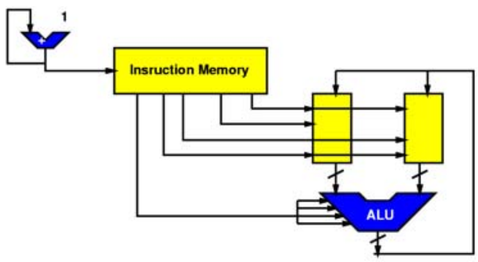
- Counter: 2
- Instruction Memory:
  000: 0000 1 001 001 010
  001: 0000 1 000 010 011
  010: 1000 1 011 100 111



- Data Memory:
  000: A
  001: B
  010: 2B
  011: A+2B
  100: 00001111
  101: ?
  110: ?
  111: ?

Third Operation

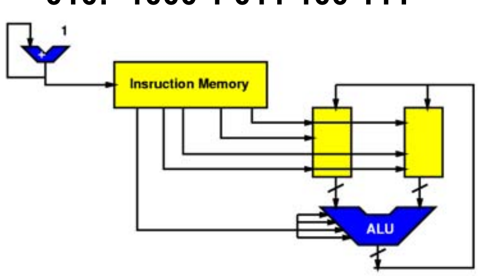
- Counter: 2
- Instruction Memory:
  000: 0000 1 001 001 010
  001: 0000 1 000 010 011
  010: 1000 1 011 100 111



- Data Memory:
  000: A
  001: B
  010: 2B
  011: A+2B
  100: 00001111
  101: ?
  110: ?
  111: ?

Third Operation Complete

- Counter: 2
- Instruction Memory:
  000: 0000 1 001 001 010
  001: 0000 1 000 010 011
  010: 1000 1 011 100 111



- Data Memory:
  000: A
  001: B
  010: 2B
  011: A+2B
  100: 00001111
  101: ?
  110: ?
  111: **(A+2B) & …**

Result

- Can sequence together primitive operations in time
- Communicating state through memory
- To perform “arbitrary” operations

“Any” Computation? (Universality)

- Any computation which can “fit” on the programmable substrate
- **Limitations**: hold entire computation and intermediate data

Motivating Questions

- What is required for recursion?
- What is the role of
  - `new`
  - `malloc`
  - `cons`

- Consider
  - routine to produce an n-element vector sum
  - downloading an image off the web
  - decompressing a downloaded file
  - read input string from user

“Any” Computation

- Computation can be of any size
- Consider UTM with unbounded input tape to describe computation

Computation Evolves During Execution

- Conventional think:
  - program graph unfolds with
    - procedure calls
    - thread spawns
  - unfold state with
    - `new`
    - `malloc`

Computing Evolves During Execution

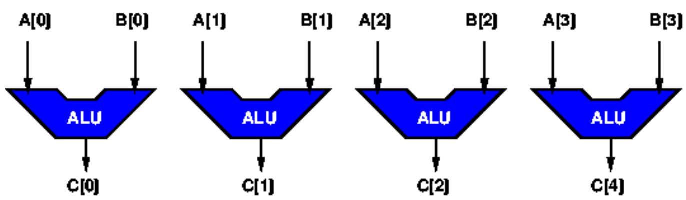
- What's happening?
  - **`new, malloc`** -- allocating new state for virtual operators
  - procedure calls and spawns -- unfolding the actual compute graph
    - from a range of possible graphs
  - use computation to **define** the computation

Example: Contrast

- Vsum(a,b)
  - c = new int[a.length()];
  - for(I=0;I<a.length();I++)
    - c[I]=a[I]+b[I];
  - return(c)
- Vsum4(a,b,c)
  - c[0]=a[0]+b[0];
  - c[1]=a[1]+b[1];
  - c[2]=a[2]+b[2];
  - c[3]=a[3]+b[3];

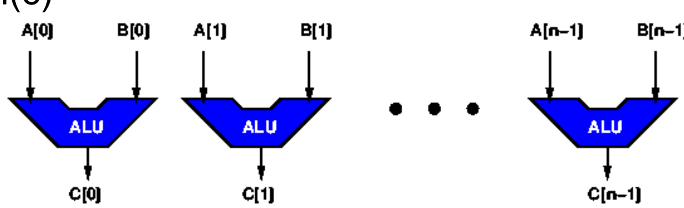
Computation: vsum4

- Vsum4(a,b,c)
  - c[0]=a[0]+b[0];
  - c[1]=a[1]+b[1];
  - c[2]=a[2]+b[2];
  - c[3]=a[3]+b[3];



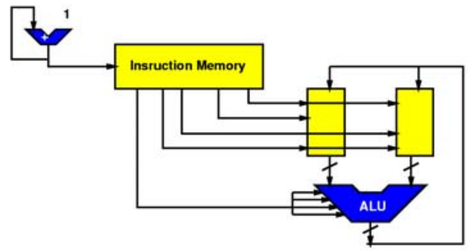
Computation: vsum

- Vsum(a,b)
  - c = new int[a.length()];
  - for(I=0;I<a.length();I++)
    - c[I]=a[I]+b[I];
  - return(c)



Compute Vsum4 on datapath

- Vsum4(a,b,c)
  - c[0]=a[0]+b[0];
  - c[1]=a[1]+b[1];
  - c[2]=a[2]+b[2];
  - c[3]=a[3]+b[3];

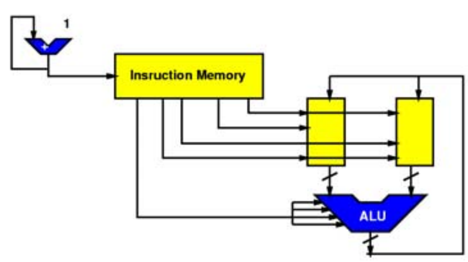


Put A's in A, B's in B
Store C's in A at end.

ADD 0,0→0
ADD 1,1→1
ADD 2,2→2
ADD 3,3→3

Compute Vsum4

- Vsum4(a,b,c)
  - c[0]=a[0]+b[0];
  - c[1]=a[1]+b[1];
  - c[2]=a[2]+b[2];
  - c[3]=a[3]+b[3];



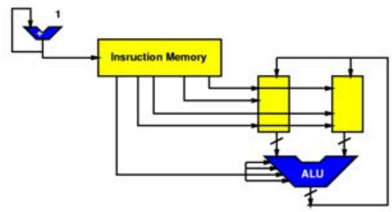
ADD 0,0→0
ADD 1,1→1
ADD 2,2→2
ADD 3,3→3

Op w src1 src2 dst

000: 0000 1 000 000 000
001: 0000 1 001 001 001
010: 0000 1 010 010 010
011: 0000 1 011 011 011

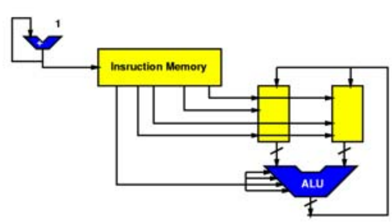
Compute Vsum

- Vsum(a,b)
  - c = new int[a.length()];
  - for(I=0;I<a.length();I++)
    - c[I]=a[I]+b[I];
  - return(c)



Compute Vsum

- Vsum(a,b)
  - c = new int[a.length()];
  - for(I=0;I<a.length();I++)
    - c[I]=a[I]+b[I];
  - return(c)

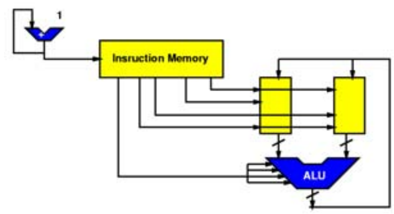


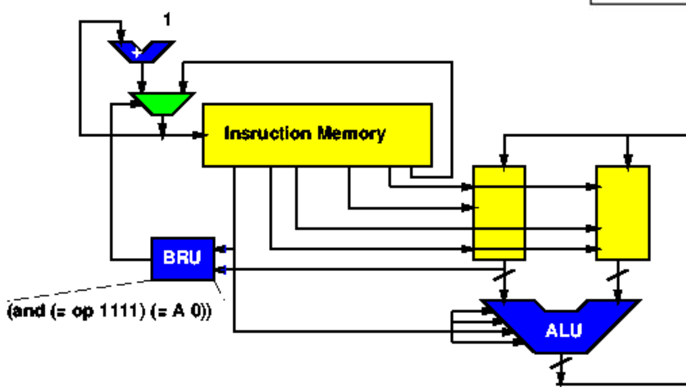
Can't do it.

•Must be able to apply operations to arbitrary data.

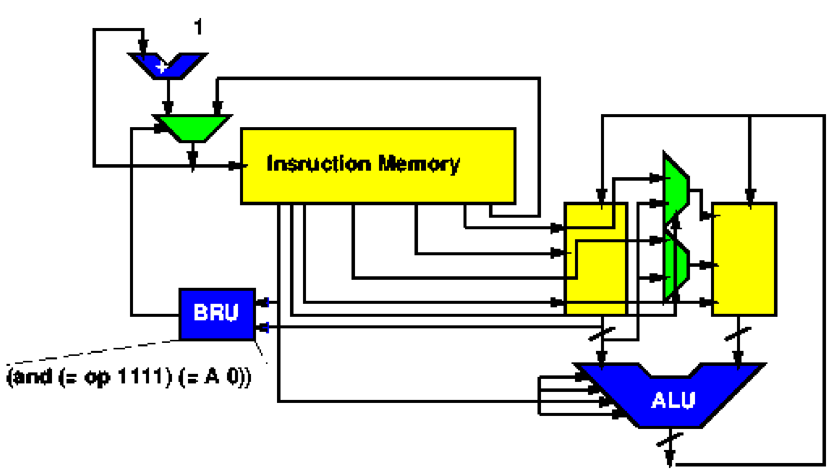
•Must run data dependent set of ops.

Add Branching



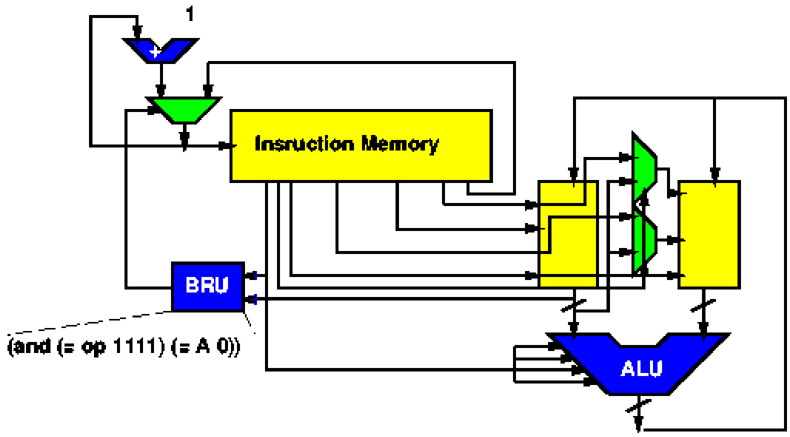


Add Data Indirect

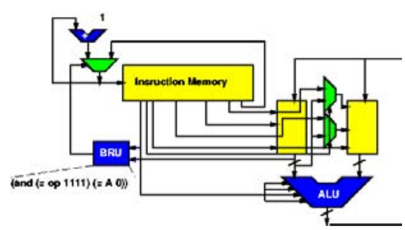


Add Data Indirect

Instr: ALUOP Bsel Write Bsrc Asrc DST Baddr



New Ops



- Important new operations:
  - DST←B[Asrc]

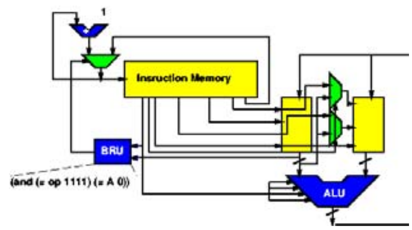
Instr: ALUOP Bsel Write Bsrc Asrc DST Baddr
B r 1 xxx Asrc DST xxx

  - B[Asrc]←Bsrc

Instr: ALUOP Bsel Write Bsrc Asrc DST Baddr
B w 1 Bsrc Asrc xxx xxx

Compute Vsum

- Vsum(a,b)
  - c = new int[a.length()];
  - for(I=0;I<a.length();I++)
    - c[I]=a[I]+b[I];
  - return(c)



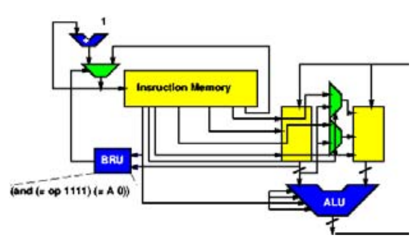
•a, b addresses in Bmem
- Values at offset 0, 1, … length
- Length at offset -1

•a, b in slots 0, 1 respectively
•Put c in slot 2

top unallocated memory in slot 3

Compute Vsum

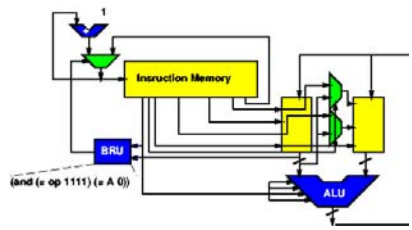
- Vsum(a,b)
  - c = new int[a.length()];
  - for(I=0;I<a.length();I++)
    - c[I]=a[I]+b[I];
  - return(c)



// allocate c
Slot 2 ← Slot 3 // start at top mem
Slot 4 ← SUB Slot 1, #1 // a-1
Slot 4 ← [Slot 4] // read a.length
Slot 3 ← Slot 3 + Slot 4 // increase
Slot 3← Slot 3 + 1 // +1 length
[Slot 2] ← Slot 4 // store length
Slot 2← Slot 2 + 1 // incr past len

Compute Vsum

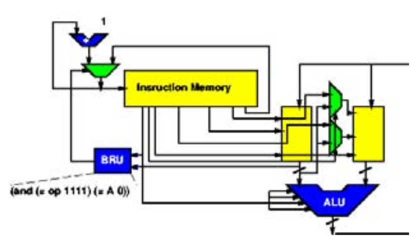
- Vsum(a,b)
  - c = new int[a.length()];
  - for(I=0;I<a.length();I++)
    - c[I]=a[I]+b[I];
  - return(c)



Plan:
4: a.length (already there)
5: i
6: cptr
7: aptr
8: bptr

Compute Vsum

- Vsum(a,b)
  - c = new int[a.length()];
  - for(I=0;I<a.length();I++)
    - c[I]=a[I]+b[I];
  - return(c)



Plan:
4: a.length
5: i
6: cptr
7: aptr
8: bptr

Slot 5←#0 // initialize I
Slot 6←Slot 2 // cptr
Slot 7←Slot 0 // aptr
Slot 8←Slot 1 // bptr

Compute Vsum

- Vsum(a,b)
  - c = new int[a.length()];
  - for(I=0;I<a.length();I++)
    - c[I]=a[I]+b[I];
  - return(c)

Plan:
4: a.length
5: i
6: cptr
7: aptr
8: bptr

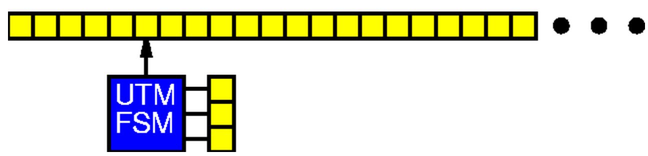
```
Loop:
  Slot 9←SUB Slot 4, Slot 5
  BRZ Slot 9 End
  Slot 10←[Slot 7] // a[I]
  Slot 11←[Slot 8] // b[I]
  Slot 10 ← Slot 10+Slot 11
  [Slot 6]← Slot 10 // c[I]
  Slot 6←Slot 6 + #1
  Slot 7←Slot 7 + #1
  Slot 8←Slot 7 + #1
  Slot 5←Slot 5 + #1
  BRZ #0  Loop:
End:
```


Memory Function

- Allow unbounded computation
- Allow computational graph to evolve during computation

Computational Strength

- With memory appropriately arranged:
  - can now compute unbounded computations
  - …but finite
- As close as we'll come to a Turing Machine



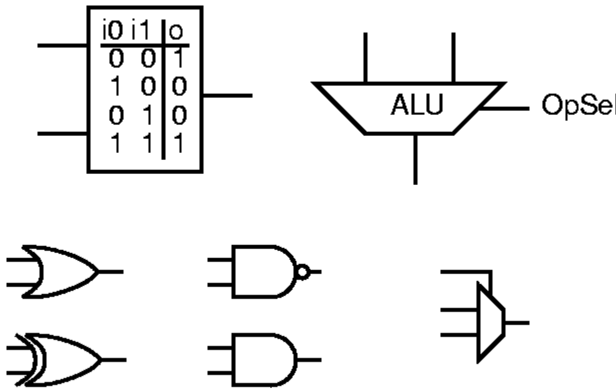
Computing Capability Review

- Gates:
  - boolean logic
  - finite functions
- Gates and registers:
  - Finite Automata
  - some infinite functions
- Memories with allocation
  - unbounded functions
  - TM w/in the limits of available memory

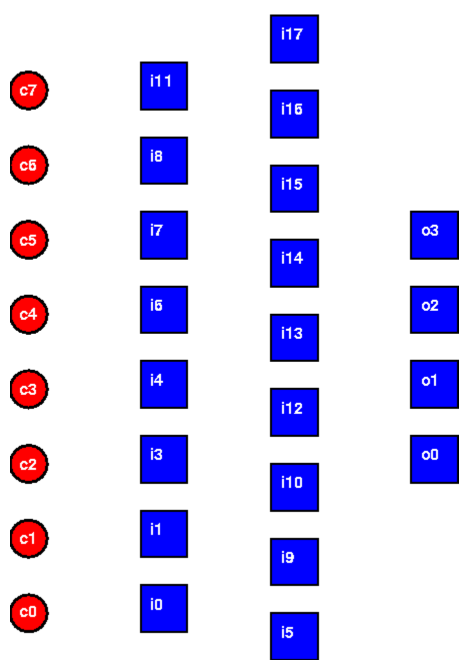
Computing Requirements (review)

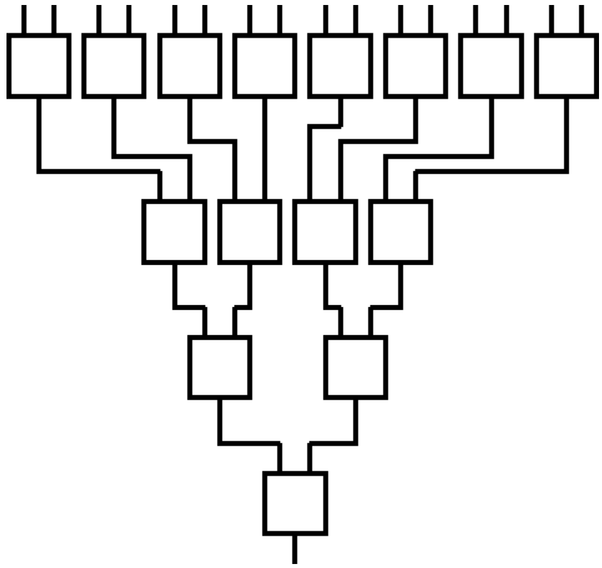
Requirements

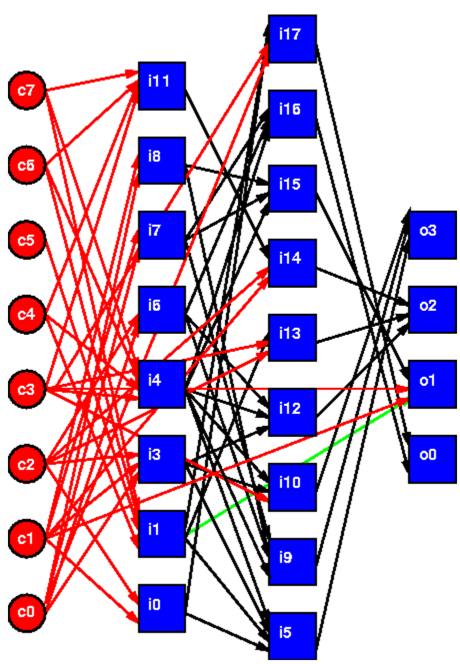
- In order to build a **general-purpose** *(programmable)* computing device, we absolutely must have?
  - _
  - _
  - _
  - _
  - _



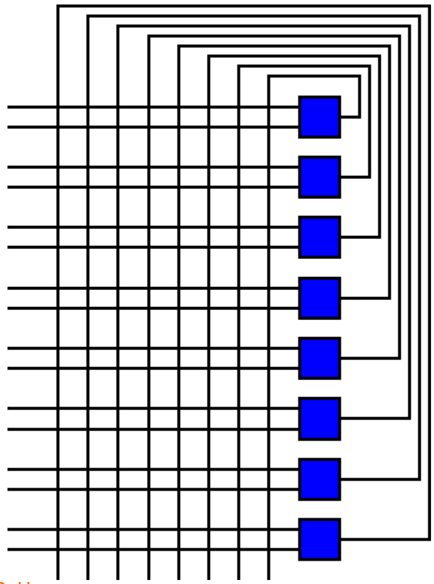
Primitive compute elements enough?



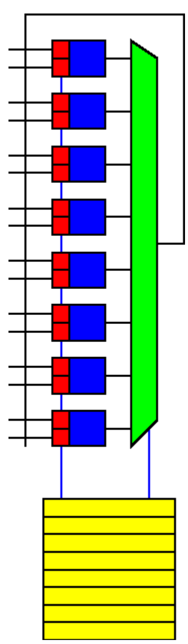



i17
i11
c7
i16
c6
i8
i15
c5
i7
o3
i14
c4
i6
o2
i13
c3
i4
o1
i12
c2
i3
o0
i10
c1
i1
i9
c0
i0
i5

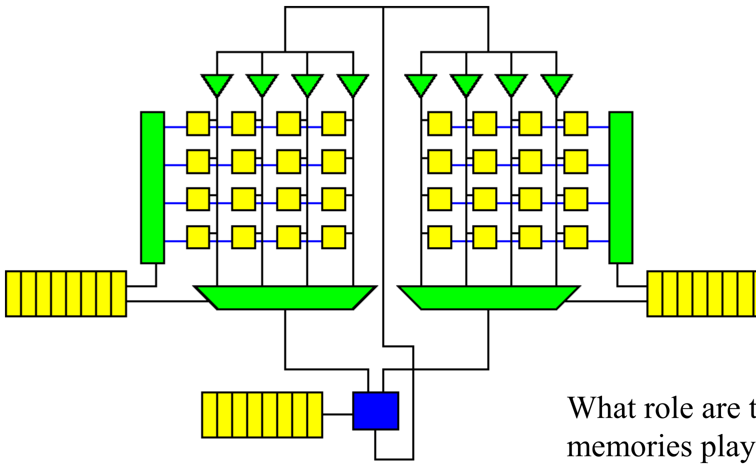
Compute and Interconnect



Sharing Interconnect Resources



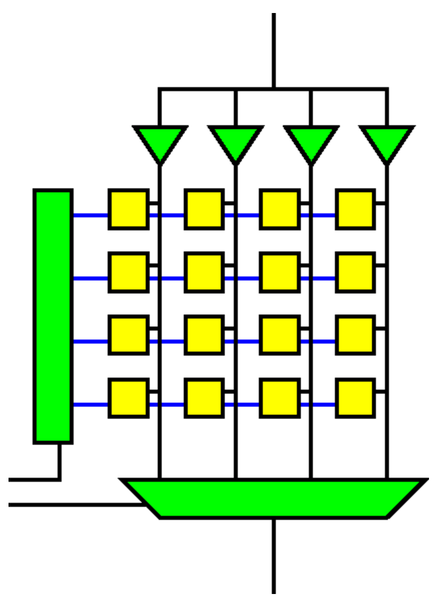
Sharing Interconnect and Compute Resources



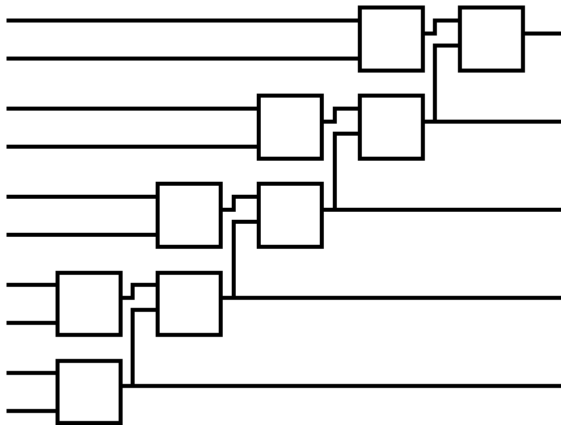
What role are the memories playing here?

Memory block or
Register File

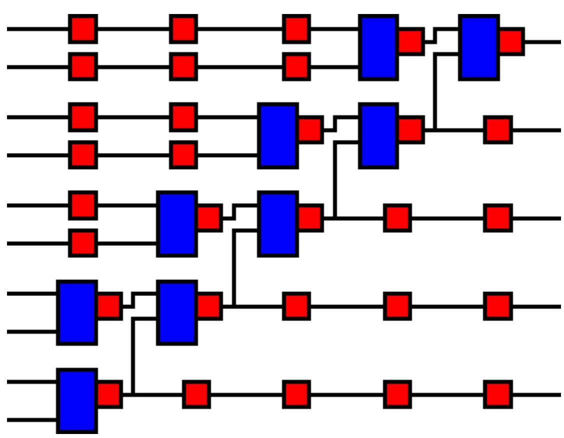
Interconnect:
moves data from input to storage cell;
or from storage cell to output.



What do I need to be able to use this circuit properly?
(reuse it on different data?)



61



62

Requirements

- In order to build a **general-purpose** *(programmable)* computing device, we absolutely must have?
  - Compute elements
  - Interconnect: space
  - Interconnect: time (retiming)
  - Interconnect: external (IO)
  - Instructions

Admin

- MLK Monday
- Next Class Wed.
  - Assignment turnin Wed. OK

Big Ideas [MSB Ideas]

- Basic elements of a programmable computation
  - Compute
  - Interconnect
    - (space and time, outside system [IO])
  - Instructions
- For unbounded computation
  - computational graph evolve/computed by computation

Big Ideas [MSB-1 Ideas]

- Two key functions of memory
  - retiming
  - instructions
    - description of computation