Controlling an ARM-like data path

Instructor: Dr. Vinicius Prado da Fonseca (vpradodafons@online.mun.ca)

Data path of arm-like

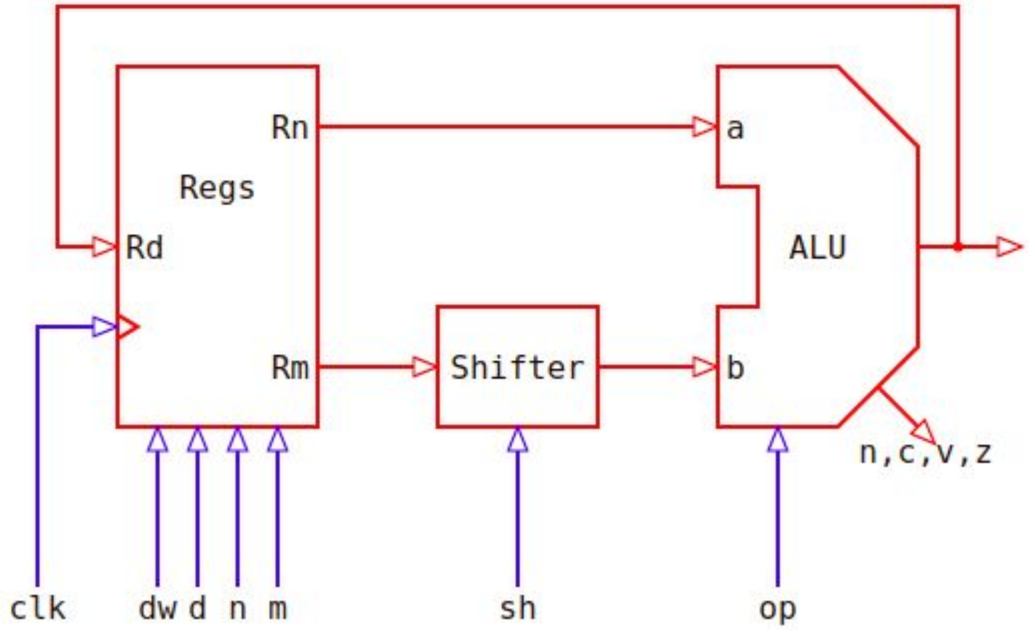
- This data path contains:
- 16-bit word 8 registers file with two read ports and one write port
  - reg16_t R[7:0]; // registers
- ALU with inputs a and b that supports
  - add
  - subtract
  - bitwise-or
  - bitwise-and
  - bitwise-xor
  - pass-b (output = operator b)
  - negate-b (invert signal)
  - invert-b (flip all bits)

typedef enum logic [2:0] {OP_ADD, OP_SUB, OP_ORR, OP_AND, OP_EOR, OP_B_NEG, OP_B_INV, OP_B_PAS} alu_ctl_t

- The shifter performs arithmetic right shift and logical left shift
  - signed [4:0] shift_amt_t

Data path of arm-like

- The block diagram is:



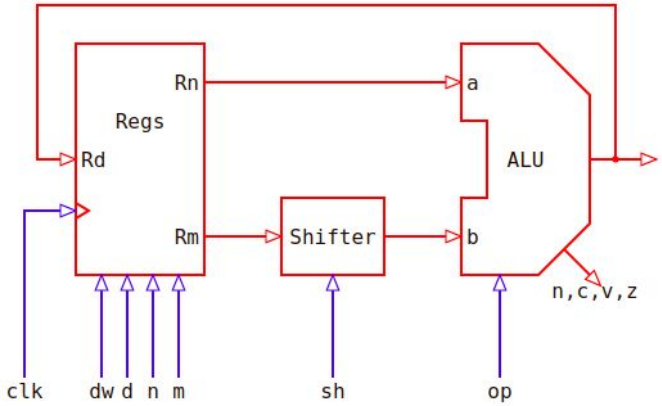
Data path of arm-like

- The data types for this data path are:

```
typedef logic [15:0] reg16_t;
typedef logic signed [4:0] shift_amt_t;
typedef logic [2:0] reg_sel_t;
typedef enum logic [2:0] {  OP_ADD, OP_SUB,
                            OP_ORR, OP_AND, OP_EOR,
                            OP_B_NEG, OP_B_INV, OP_B_PAS
                          } alu_ctl_t;
```

Data path of arm-like

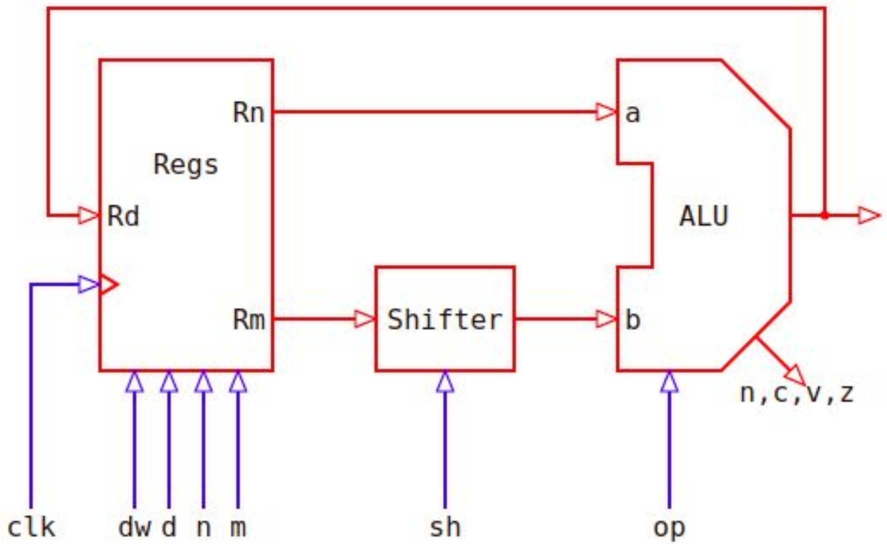
- armish_dp.v
- “shifter” module implements the combinational block that does right and left shift operations
- “arm_alu” implements the ALU
- The register file is implemented with reg_file
- The modules are combined to create the data path with the “armish_datapath” module



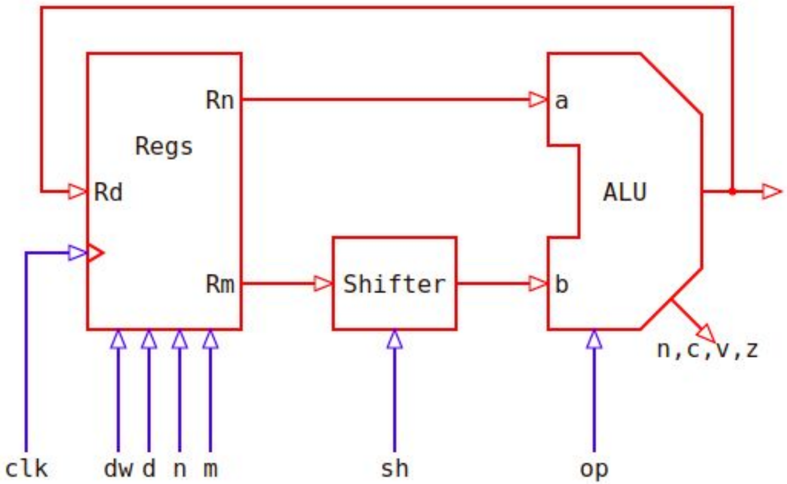
Data path of arm-like

The ports of the armish_datapath provide:

- outputs:
  - out - Output from the ALU (reg16_t)
  - cn - N (negative)
  - cz - Z (zero)
  - cc - C (carry out)
  - cv - V (overflow)
  - all operations set the flags
    - check the notes



Data path of arm-like



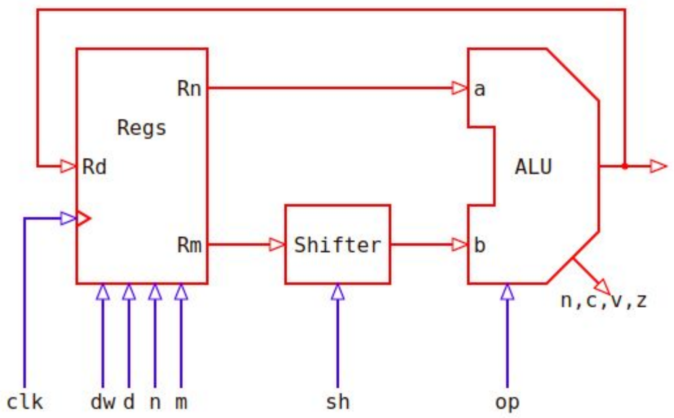
The ports of the armish_datapath provide:

- inputs:
  - op - specifies one of the operations (alu_ctl_t)
  - sh (signed) - controls the shift amount (shift_amt_t)
    - positive for left shifts
    - negative for arithmetic right shifts
  - d - selects the destination register that is updated (reg_sel_t)
  - n, m - register reading location
  - dw, clk, reset
    - dw enables register file updates. dw is 1, the output of the ALU is stored
    - clk is the clock signal for the positive edge flip-flops
    - reset clears all the flip-flops to zero.

RTL Description of the data path

- The behaviour of this data path can be analysed at the register transfer level

  R[0] <= R[1] + R[5]
  dw=?; d=?; n=?; m=?; sh=?; op=?

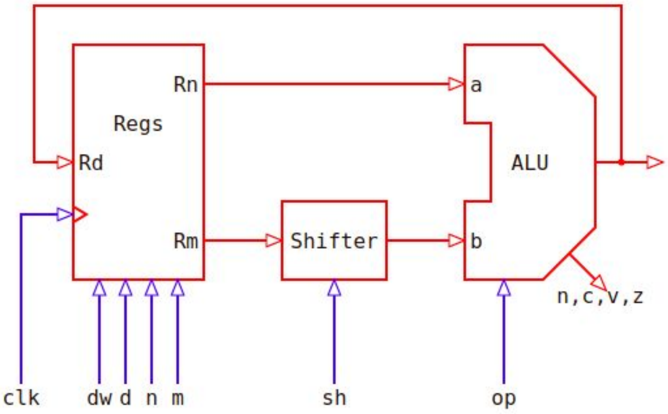


dw	d	n	m	sh	op	RTL
1	x	-	z	0	pass_b	R[x] <= R[z]
1	x	-	z	0	invert_b	R[x] <= ~R[z]
1	x	-	z	0	negate_b	R[x] <= -R[z]
1	x	y	z	0	or	R[x] <= R[y] | R[z]
1	x	y	z	0	eor	R[x] <= R[y] ^ R[z]
1	x	y	z	0	and	R[x] <= R[y] & R[z]
1	x	y	z	0	add	R[x] <= R[y] + R[z]
1	x	y	z	0	sub	R[x] <= R[y] - R[z]
0	-	y	z	0	sub	R[y] - R[z] for compare
1	x	y	z	sh	op	R[x] <= R[y] op (R[z] << sh) sh is positive
1	x	y	z	sh	op	R[x] <= R[y] op (R[z] >>> -sh) sh is negative

RTL Description of the data path

- The behaviour of this data path can be analysed at the register transfer level

  R[0] <= R[1] + R[5]
  dw=1; d=0; n=1; m=5; sh=0; op=OP_ADD

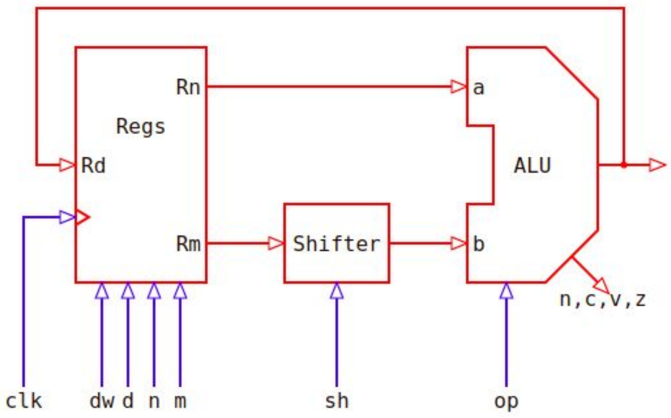


dw	d	n	m	sh	op	RTL
1	x	-	z	0	pass_b	R[x] <= R[z]
1	x	-	z	0	invert_b	R[x] <= ~R[z]
1	x	-	z	0	negate_b	R[x] <= -R[z]
1	x	y	z	0	or	R[x] <= R[y] | R[z]
1	x	y	z	0	eor	R[x] <= R[y] ^ R[z]
1	x	y	z	0	and	R[x] <= R[y] & R[z]
1	x	y	z	0	add	R[x] <= R[y] + R[z]
1	x	y	z	0	sub	R[x] <= R[y] - R[z]
0	-	y	z	0	sub	R[y] - R[z] for compare
1	x	y	z	sh	op	R[x] <= R[y] op (R[z] << sh) sh is positive
1	x	y	z	sh	op	R[x] <= R[y] op (R[z] >>> -sh) sh is negative

RTL Description of the data path

- sh and alu in the same clock?
  R[1] <=R[2] + R[5] << 1
  R1 = R2 + R5*2

dw=1; d=1; n=2; m=5; sh=1; op=OP_ADD

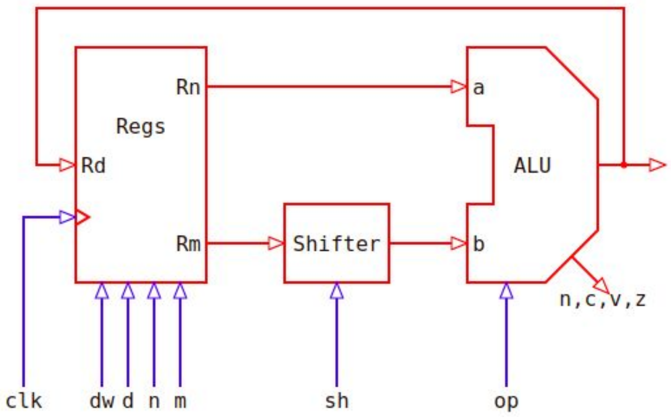


dw	d	n	m	sh	op	RTL
1	x	-	z	0	pass_b	R[x] <= R[z]
1	x	-	z	0	invert_b	R[x] <= ~R[z]
1	x	-	z	0	negate_b	R[x] <= -R[z]
1	x	y	z	0	or	R[x] <= R[y] | R[z]
1	x	y	z	0	eor	R[x] <= R[y] ^ R[z]
1	x	y	z	0	and	R[x] <= R[y] & R[z]
1	x	y	z	0	add	R[x] <= R[y] + R[z]
1	x	y	z	0	sub	R[x] <= R[y] - R[z]
0	-	y	z	0	sub	R[y] - R[z] for compare
1	x	y	z	sh	op	R[x] <= R[y] op (R[z] << sh) sh is positive
1	x	y	z	sh	op	R[x] <= R[y] op (R[z] >>> -sh) sh is negative

RTL Description of the data path

- Only shifting?
  R[1] <=R[5] << 1 (operand b, m)
  R1 = R5*2

dw=1; d=1; n=x; m=5; sh=1; op=OP_PASS_B

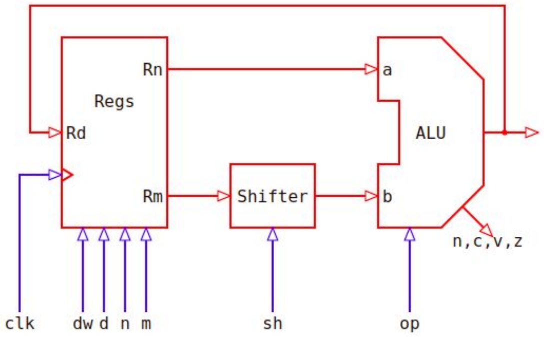


dw	d	n	m	sh	op	RTL
1	x	-	z	0	pass_b	R[x] <= R[z]
1	x	-	z	0	invert_b	R[x] <= ~R[z]
1	x	-	z	0	negate_b	R[x] <= -R[z]
1	x	y	z	0	or	R[x] <= R[y] | R[z]
1	x	y	z	0	eor	R[x] <= R[y] ^ R[z]
1	x	y	z	0	and	R[x] <= R[y] & R[z]
1	x	y	z	0	add	R[x] <= R[y] + R[z]
1	x	y	z	0	sub	R[x] <= R[y] - R[z]
0	-	y	z	0	sub	R[y] - R[z] for compare
1	x	y	z	sh	op	R[x] <= R[y] op (R[z] << sh) sh is positive
1	x	y	z	sh	op	R[x] <= R[y] op (R[z] >>> -sh) sh is negative

RTL Description of the data path

- The behaviour of this data path can be analysed at the register transfer level
- sh and alu in the same clock

How to set register 3 to 0?

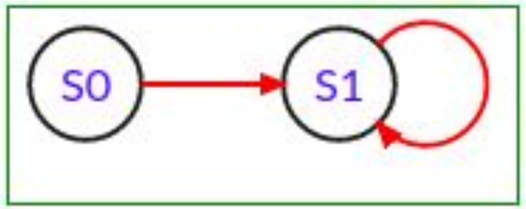


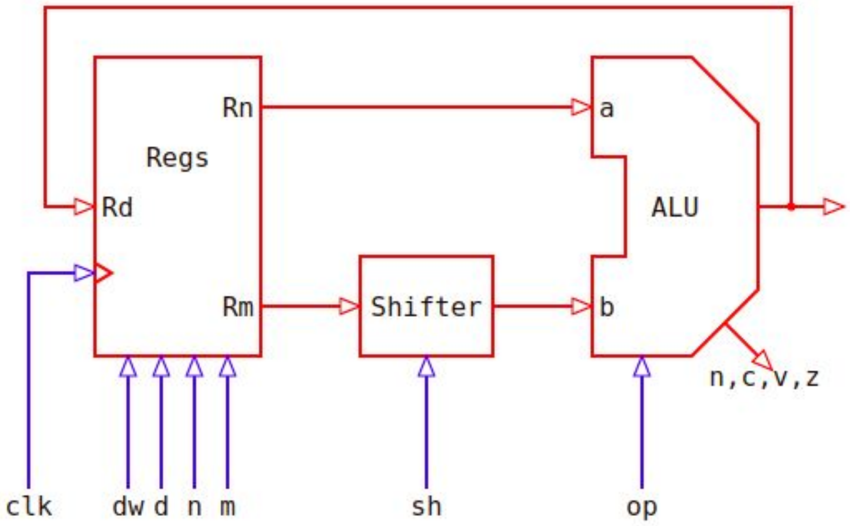
dw	d	n	m	sh	op	RTL
1	x	-	z	0	pass_b	R[x] <= R[z]
1	x	-	z	0	invert_b	R[x] <= ~R[z]
1	x	-	z	0	negate_b	R[x] <= -R[z]
1	x	y	z	0	or	R[x] <= R[y] | R[z]
1	x	y	z	0	eor	R[x] <= R[y] ^ R[z]
1	x	y	z	0	and	R[x] <= R[y] & R[z]
1	x	y	z	0	add	R[x] <= R[y] + R[z]
1	x	y	z	0	sub	R[x] <= R[y] - R[z]
0	-	y	z	0	sub	R[y] - R[z] for compare
1	x	y	z	sh	op	R[x] <= R[y] op (R[z] << sh) sh is positive
1	x	y	z	sh	op	R[x] <= R[y] op (R[z] >>> -sh) sh is negative

Controller for R[3] <= 0

- Register 3, R[3], can be initialized to 0 with:

```
S0: R[3] <= R[3] ^ R[3]

op=OP_EOR, sh=0, d=3, n=3, m=3, dw=1
```

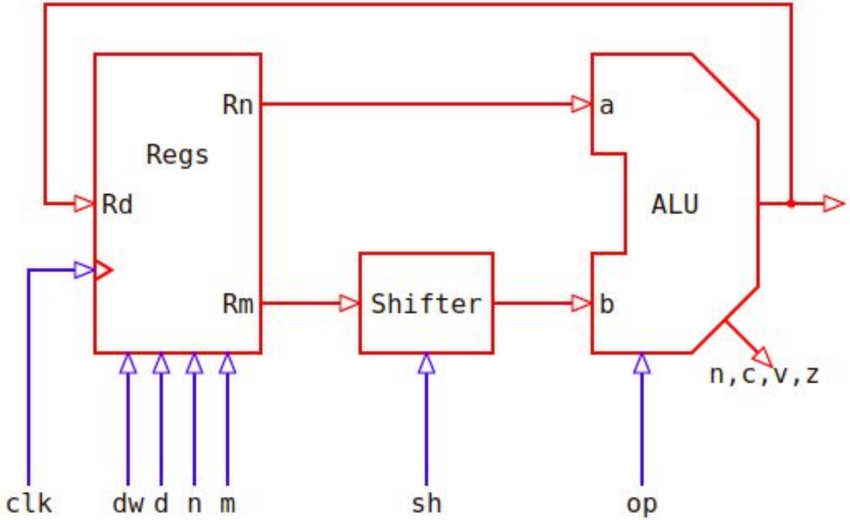




Controller for R[3] <= 0

- Register 3, R[3], can be initialized to 0 with:

```
S0: R[3] <= R[3] ^ R[3]

op=OP_EOR, sh=0, d=3, n=3, m=3, dw=1

S1: pass_b(R[3])

op=OP_B_PAS, sh=0, d=0, n=0, m=3, dw=0
```



Controller for R[3] <= 0

- Register 3, R[3], can be initialized to 0 with:

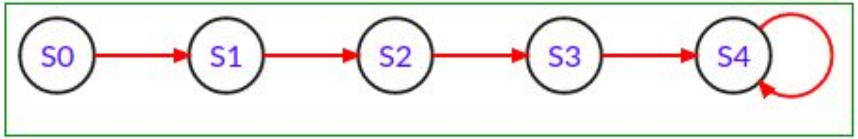
```
S0: R[3] <= R[3] ^ R[3]

op=OP_EOR, sh=0, d=3, n=3, m=3, dw=1

S1: pass_b(R[3])

op=OP_B_PAS, sh=0, d=0, n=0, m=3, dw=0
```

- S1 is used to output R[3] through the ALU
- Transitions similar to mini datapath

Controller for R[6] <= 1

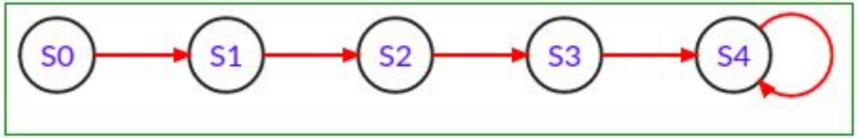
Register 6, R[6], can be initialized to 1 with:

dw	d	n	m	sh	op	RTL
1	x	-	z	0	pass_b	R[x] <= R[z]
1	x	-	z	0	invert_b	R[x] <= ~R[z]
1	x	-	z	0	negate_b	R[x] <= -R[z]
1	x	y	z	0	or	R[x] <= R[y] | R[z]
1	x	y	z	0	eor	R[x] <= R[y] ^ R[z]
1	x	y	z	0	and	R[x] <= R[y] & R[z]
1	x	y	z	0	add	R[x] <= R[y] + R[z]
1	x	y	z	0	sub	R[x] <= R[y] - R[z]
0	-	y	z	0	sub	R[y] - R[z] for compare
1	x	y	z	sh	op	R[x] <= R[y] op (R[z] << sh) sh is positive
1	x	y	z	sh	op	R[x] <= R[y] op (R[z] >>> -sh) sh is negative



Controller for R[6] `<= 1`

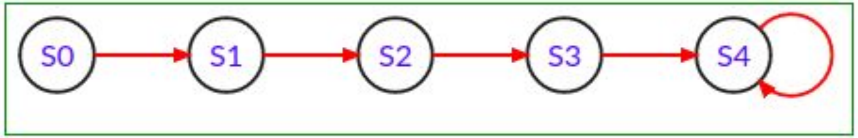
Register 6, R[6], can be initialized to 1 with:




```
S0: R[6] <= R[6] ^ R[6]   // R[6] = 0000 0000 0000 0000
```

Controller for R[6] `<= 1`

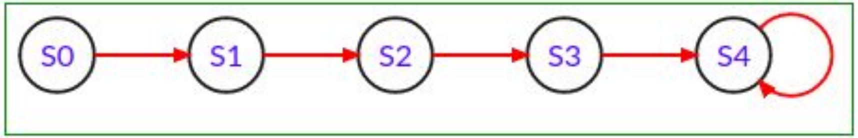
Register 6, R[6], can be initialized to 1 with:




```
S0: R[6] <= R[6] ^ R[6]    // R[6] = 0000 0000 0000 0000

S1: R[6] <= ~R[6]          // R[6] = 1111 1111 1111 1111
```

Controller for R[6] <= 1

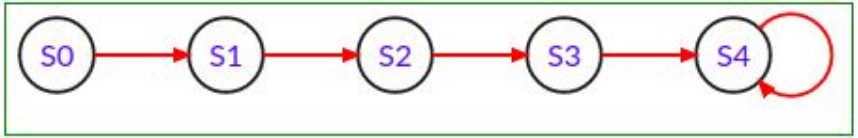
Register 6, R[6], can be initialized to 1 with:




```
S0: R[6] <= R[6] ^ R[6]   // R[6] = 0000 0000 0000 0000

S1: R[6] <= ~R[6]         // R[6] = 1111 1111 1111 1111

S2: R[6] <= R[6] << 1     // R[6] = 1111 1111 1111 1110 (sh = 00001)
```

Controller for R[6] `<= 1`

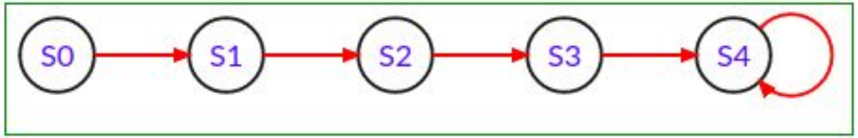
Register 6, R[6], can be initialized to 1 with:




```
S0: R[6] <= R[6] ^ R[6]   // R[6] = 0000 0000 0000 0000

S1: R[6] <= ~R[6]         // R[6] = 1111 1111 1111 1111

S2: R[6] <= R[6] << 1     // R[6] = 1111 1111 1111 1110 (sh = 00001)

S3: R[6] <= ~R[6]         // R[6] = 0000 0000 0000 0001
```

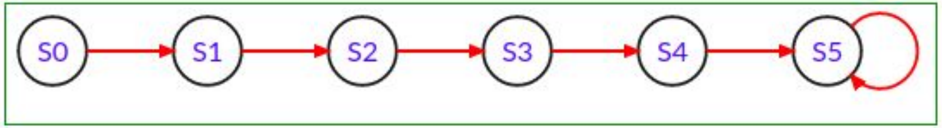
Controller for R[6] <= 1

Register 6, R[6], can be initialized to 1 with:




```
S0: R[6] <= R[6] ^ R[6]   // R[6] = 0000 0000 0000 0000

S1: R[6] <= ~R[6]         // R[6] = 1111 1111 1111 1111

S2: R[6] <= R[6] << 1     // R[6] = 1111 1111 1111 1110 (sh = 00001)

S3: R[6] <= ~R[6]         // R[6] = 0000 0000 0000 0001

S4: pass_b(R[6])
```

Controller for R[4] <= 32 (hint: shifting)

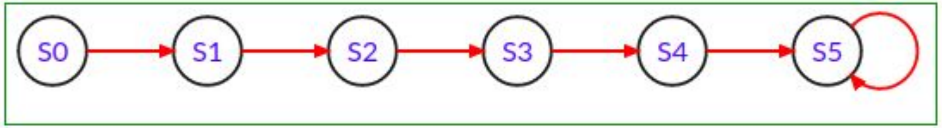


R[4] can be initialized to 32 with:

```
S0: R[4] <= R[4] ^ R[4]   // R[4] = 0000 0000 0000 0000
```

Controller for R[4] <= 32

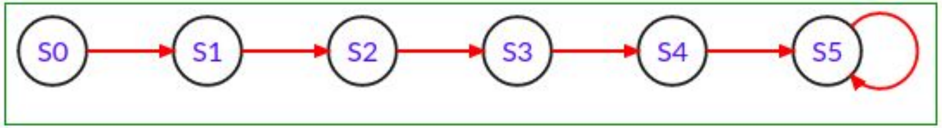
R[4] can be initialized to 32 with:




```
S0: R[4] <= R[4] ^ R[4]   // R[4] = 0000 0000 0000 0000

S1: R[4] <= ~R[4]         // R[4] = 1111 1111 1111 1111
```

Controller for R[4] <= 32

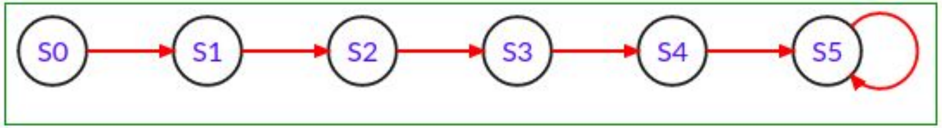
R[4] can be initialized to 32 with:




```
S0: R[4] <= R[4] ^ R[4]   // R[4] = 0000 0000 0000 0000

S1: R[4] <= ~R[4]         // R[4] = 1111 1111 1111 1111

S2: R[4] <= R[4] << 1     // R[4] = 1111 1111 1111 1110 (sh=00001)
```

Controller for R[4] <= 32

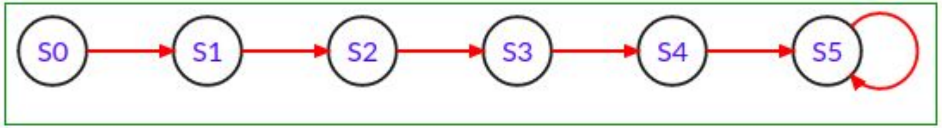
R[4] can be initialized to 32 with:




```
S0: R[4] <= R[4] ^ R[4]    // R[4] = 0000 0000 0000 0000

S1: R[4] <= ~R[4]          // R[4] = 1111 1111 1111 1111

S2: R[4] <= R[4] << 1      // R[4] = 1111 1111 1111 1110 (sh=00001)

S3: R[4] <= ~R[4]          // R[4] = 0000 0000 0000 0001
```

Controller for R[4] <= 32

R[4] can be initialized to 32 with:

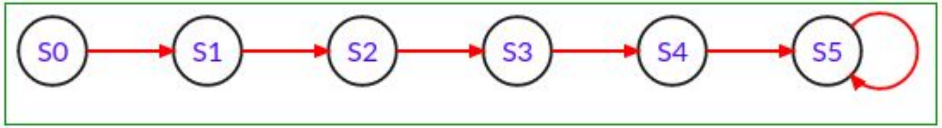



```
S0: R[4] <= R[4] ^ R[4]   // R[4] = 0000 0000 0000 0000

S1: R[4] <= ~R[4]         // R[4] = 1111 1111 1111 1111

S2: R[4] <= R[4] << 1     // R[4] = 1111 1111 1111 1110 (sh=00001)

S3: R[4] <= ~R[4]         // R[4] = 0000 0000 0000 0001

S4: R[4] <= R[4] << 5     // R[4] = 0000 0000 0010 0000 (sh=00101)
```


MEMORIAL UNIVERSITY
www.mun.ca

Controller for R[4] <= 32

R[4] can be initialized to 32 with:




```
S0: R[4] <= R[4] ^ R[4]   // R[4] = 0000 0000 0000 0000

S1: R[4] <= ~R[4]         // R[4] = 1111 1111 1111 1111

S2: R[4] <= R[4] << 1     // R[4] = 1111 1111 1111 1110 (sh=00001)

S3: R[4] <= ~R[4]         // R[4] = 0000 0000 0000 0001

S4: R[4] <= R[4] << 5     // R[4] = 0000 0000 0010 0000 (sh=00101)

S5: pass_b(R[4])
```

- The first four states are, almost identical R[6] = 1

Controller for maximum of R[0], R[1], R[2]

A Python script that finds the maximum of r0, r1, and r2 is:

```
def max( r0, r1, r2):

    if ( r0 < r1 ):

        r0 = r1

    if ( r0 < r2 ):

        r0 = r2

    return r0
```

Controller for maximum of R[0], R[1], R[2]

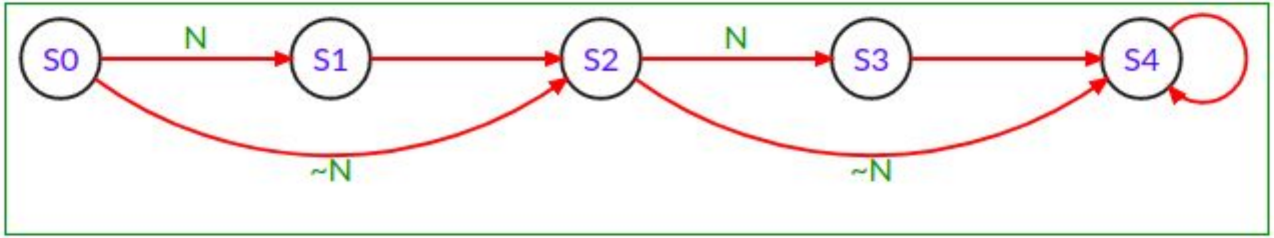
- A RTL description for finding maximum is:

```
S0: R[0] - R[1]            // compare R[0] and R[1] (OP_SUB, dw=0)

S1: R[0] <= R[1]           // R[1] is larger

S2: R[0] - R[2]            // compare R[0] and R[2] (OP_SUB, dw=0)

S3: R[0] <= R[2]           // R[2] is larger

S4: pass_b(R[0])           // output R[0]
```

- Transitions use cn (Negative) to skip states and perform the operation correctly

Controller for maximum of R[0], R[1], R[2]

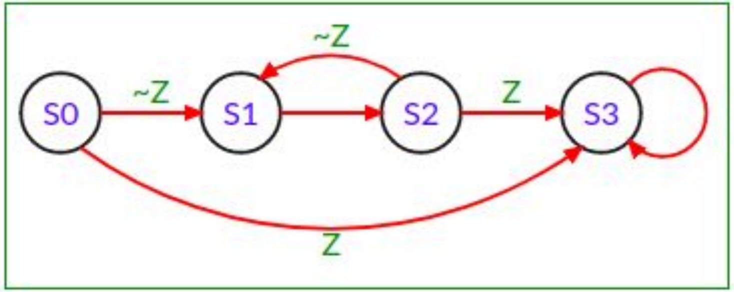
```
case( st )
     S0: st <= cn ? S1 : S2;
     S1: st <= S2;
     S2: st <= cn ? S3 : S4;
     S3: st <= S4;
     S4: st <= S4;
endcase
```



Controller for summing n numbers

This controller is equivalent to the following Python script. Add numbers from n to zero.

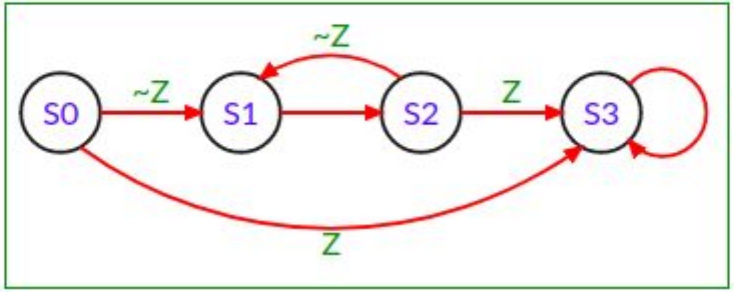
```
n = 4    # R[0]

sum = 0 # R[2]

one = 1 # R[1]

while n > 0 :

    sum = sum + n

    n = n - one
```



Controller for summing n numbers

R[0] = n, R[1] = 1, and R[2] = 0

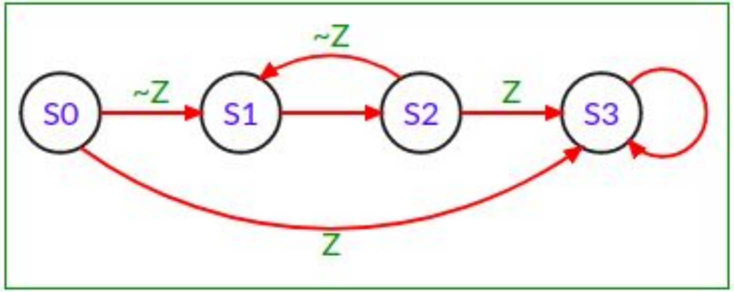
```
S0: R[0] | R[0]          // check if R[0] is 0

S1: R[2] <= R[2] + R[0]   // accumulate sum

S2: R[0] <= R[0] - R[1]   // n = n - 1, check for 0

S3: pass_b(R[2])          // output R[2]
```



Controller for summing n numbers

S0 and S2 go to the final state S3, if the data path's zero condition is true (cz = 1)

```
S0: st <= cz ? S3 : S1;

S1: st <= S2;

S2: st <= cz ? S3 : S1;

S3: st <= S3;
```



- Transitions use cz (Zero) to loop back to state S1 while R0 != 0

Controller for Euclid's algorithm

Euclid's algorithm is used to calculate the greatest common divisor

a=12, b=4

12 > 4

a = 12 - 4, a = 8

8 > 4

a = 8 - 4, a = 4

a == b, return 4

A Python's version is

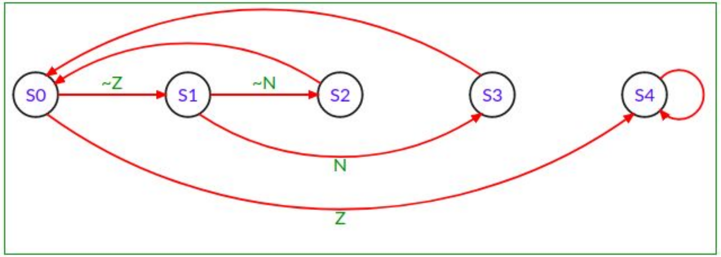
```
def gcd( a, b) :
    while a != b :
        if a > b :
            a = a-b
        else :
            b = b-a
    return a
```

Controller for Euclid's algorithm

R[3] = a, and R[4]=b, an RTL implementation for Euclid's algorithm is:

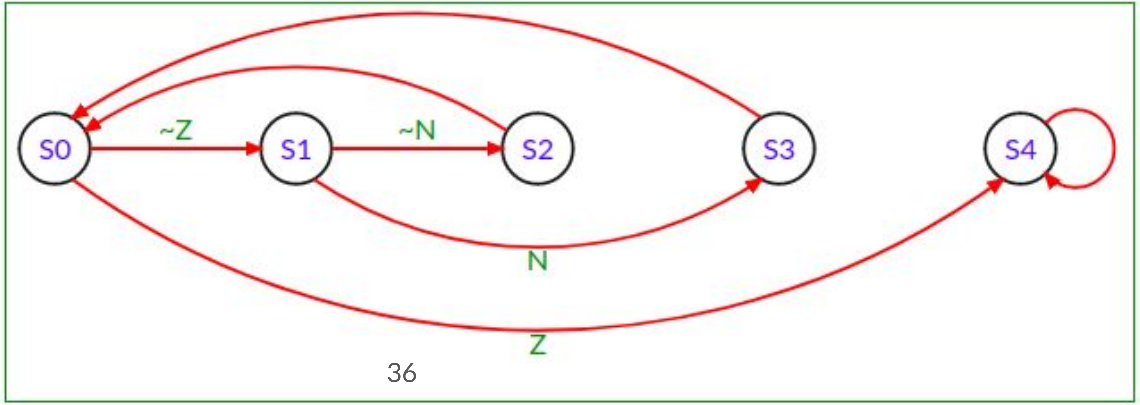
```
S0: R[0] <= R[3] - R[4]        // check if a == b

S1: R[0] | R[0]                          // check if a-b is negative

S2: R[3] <= R[3] - R[4]        // a = a - b

S3: R[4] <= R[4] - R[3]        // b = b - a

S4: pass_b(R[3])                         // output R[3]
```

```
Python:

def gcd( a, b) :

      while a != b :

            if a > b :

                  a = a-b

            else :

                  b = b-a

      return a
```



Controller for Euclid's algorithm

```
S0: st <= cz ? S4 : S1; // a==b?

S1: st <= cn ? S3 : S2; // a<b?

S2: st <= S0;

S3: st <= S0;

S4: st <= S4;
```



Next steps

Examples in the notes

Run and modify the examples