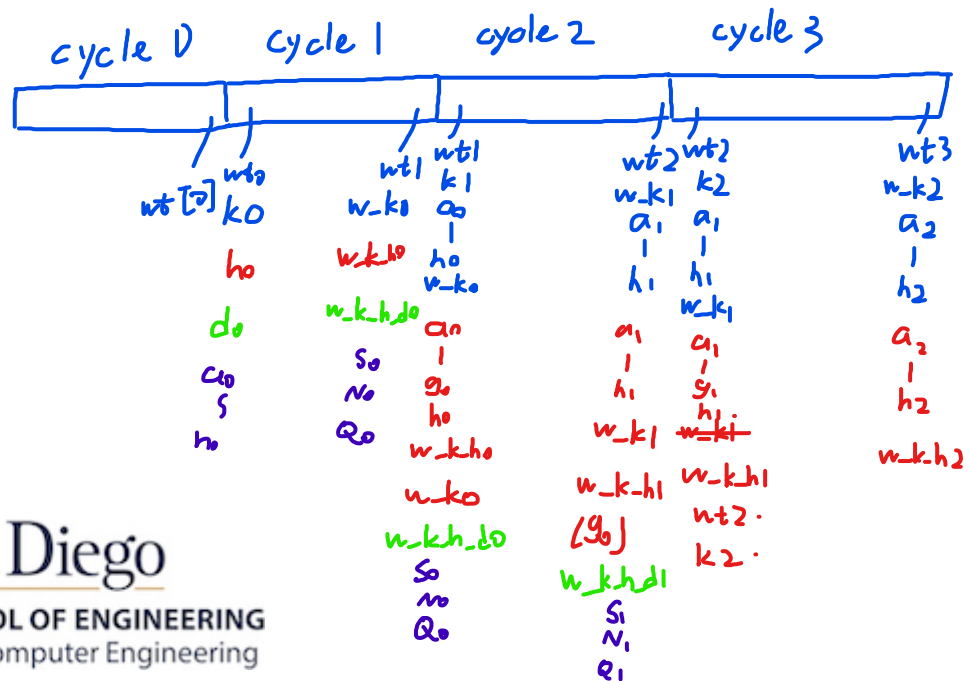
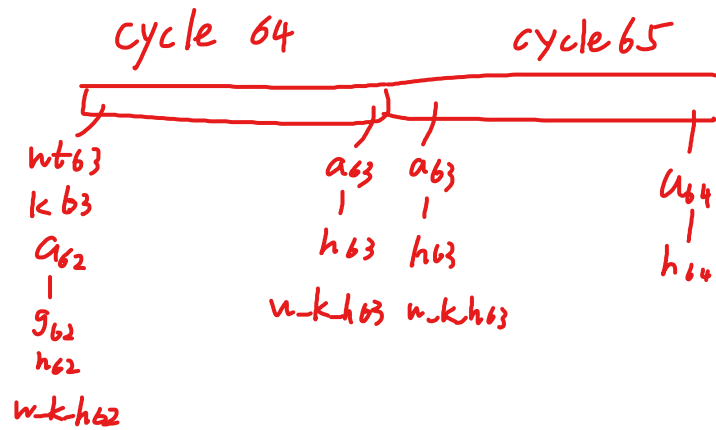


SHA256 Optimization





Optimizing SHA256

Each SHA256 Round

- There is really only one set of {A, B, C, D, E, F, G, H} registers.

$$S_0 = (A \text{ rightrotate } 2) \text{ xor } (A \text{ rightrotate } 13) \text{ xor } (A \text{ rightrotate } 22)$$

$$\text{maj} = (A \text{ and } B) \text{ xor } (A \text{ and } C) \text{ xor } (B \text{ and } C)$$

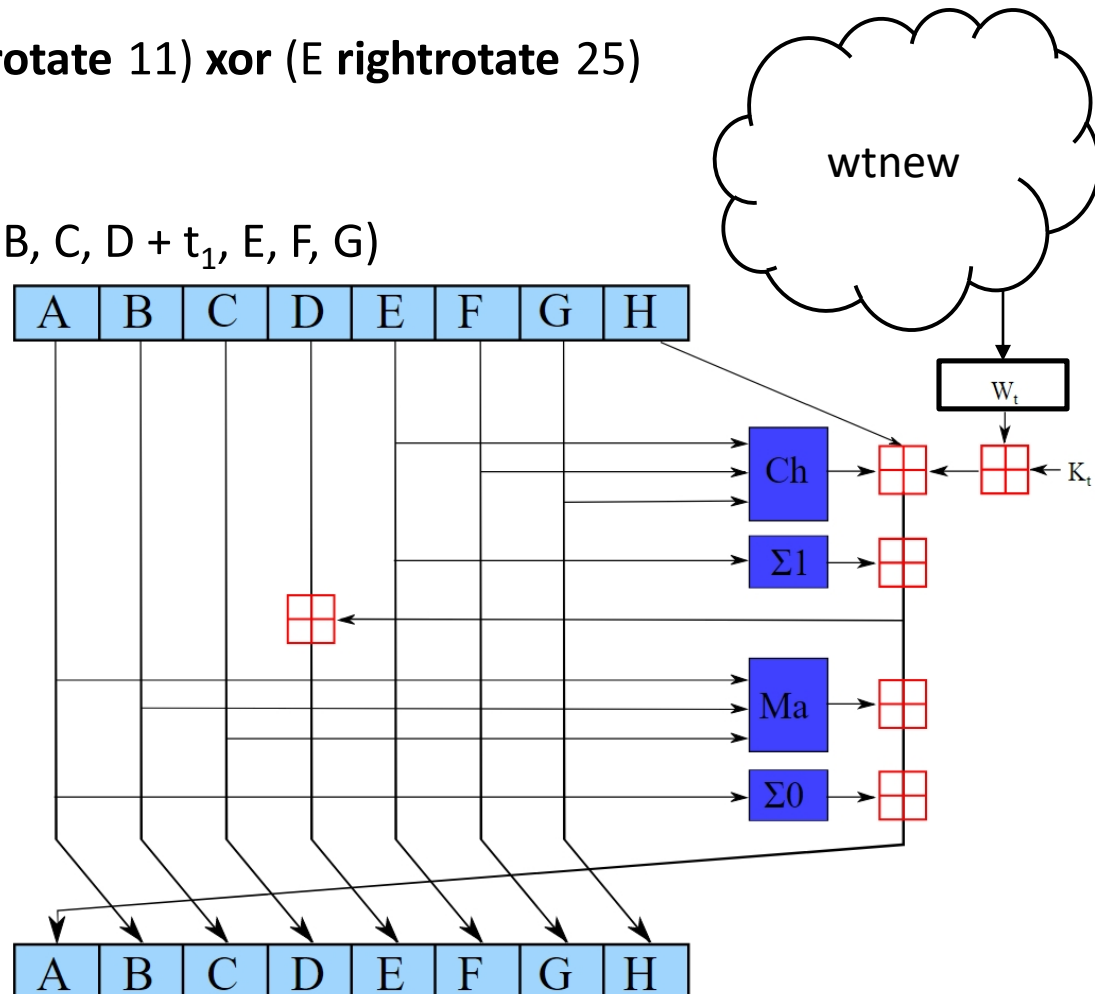
$$t_2 = S_0 + \text{maj}$$

$$S_1 = (E \text{ rightrotate } 6) \text{ xor } (E \text{ rightrotate } 11) \text{ xor } (E \text{ rightrotate } 25)$$

$$\text{ch} = (E \text{ and } F) \text{ xor } ((\text{not } E) \text{ and } G)$$

$$t_1 = H + S_1 + \text{ch} + K_t + W_t$$

$$(A, B, C, D, E, F, G, H) = (t_1 + t_2, A, B, C, D + t_1, E, F, G)$$



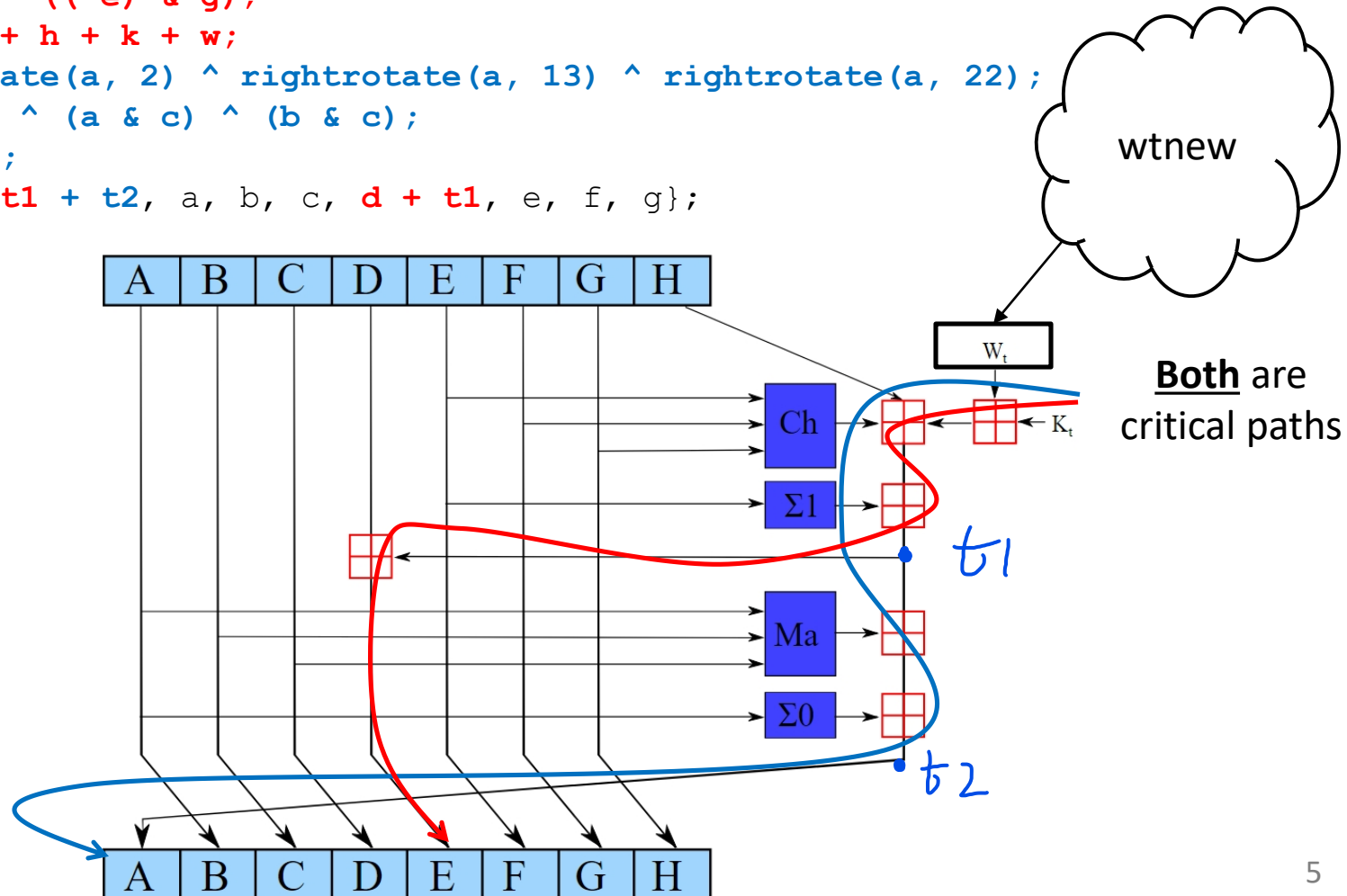
SHA256 logic

```
function logic [255:0] sha256_op(input logic [31:0] a, b, c, d, e, f, g, h, w, k);
    logic [31:0] S1, S0, ch, maj, t1, t2; // internal signals
begin
    S1 = rightrotate(e, 6) ^ rightrotate(e, 11) ^ rightrotate(e, 25);
    ch = (e & f) ^ ((~e) & g);
    t1 = ch + S1 + h + k + w;
    S0 = rightrotate(a, 2) ^ rightrotate(a, 13) ^ rightrotate(a, 22);
    maj = (a & b) ^ (a & c) ^ (b & c);
    t2 = maj + S0;
    sha256_op = {t1 + t2, a, b, c, d + t1, e, f, g};
end
endfunction

always_ff @(...) begin
    if (!reset_n) begin
        ...
    end else case(state)
        ...
        COMPUTE: begin
            ...
            {a, b, c, d, e, f, g, h} <= sha256_op(a, b, c, d, e, f, g, h, w, k[t]);
            ...
        end
        ...
    endcase
end
```

Critical Path

```
function logic [255:0] sha256_op(input logic [31:0] a, b, c, d, e, f, g, h, w, k);
    logic [31:0] S1, S0, ch, maj, t1, t2; // internal signals
begin
    S1 = rightrotate(e, 6) ^ rightrotate(e, 11) ^ rightrotate(e, 25);
    ch = (e & f) ^ ((~e) & g);
    t1 = ch + S1 + h + k + w;
    S0 = rightrotate(a, 2) ^ rightrotate(a, 13) ^ rightrotate(a, 22);
    maj = (a & b) ^ (a & c) ^ (b & c);
    t2 = maj + S0;
    sha256_op = {t1 + t2, a, b, c, d + t1, e, f, g};
end
endfunction
```



Hints for $W[n]$ array

- For $16 \leq t \leq 63$

$$s_0 = (W_{t-15} \text{ rightrotate } 7) \text{ xor } (W_{t-15} \text{ rightrotate } 18) \text{ xor } (W_{t-15} \text{ rightshift } 3)$$

$$s_1 = (W_{t-2} \text{ rightrotate } 17) \text{ xor } (W_{t-2} \text{ rightrotate } 19) \text{ xor } (W_{t-2} \text{ rightshift } 10)$$

$$W_t = W_{t-16} + s_0 + W_{t-7} + s_1$$

- A straightforward way to implement SHA256 is to use an array of **64** 32-bit words to implement W_t

```
logic [31:0] w[64];
```

then compute a new W_t as follows:

```
function logic [31:0] wtnew; // function with no inputs
    logic [31:0] s0, s1;
```

```
    s0 = rrot(w[t-15],7)^rrot(w[t-15],18)^(w[t-15]>>3);
```

```
    s1 = rrot(w[t-2],17)^rrot(w[t-2],19)^(w[t-2]>>10);
```

```
    wtnew = w[t-16] + s0 + w[t-7] + s1;
```

```
endfunction
```

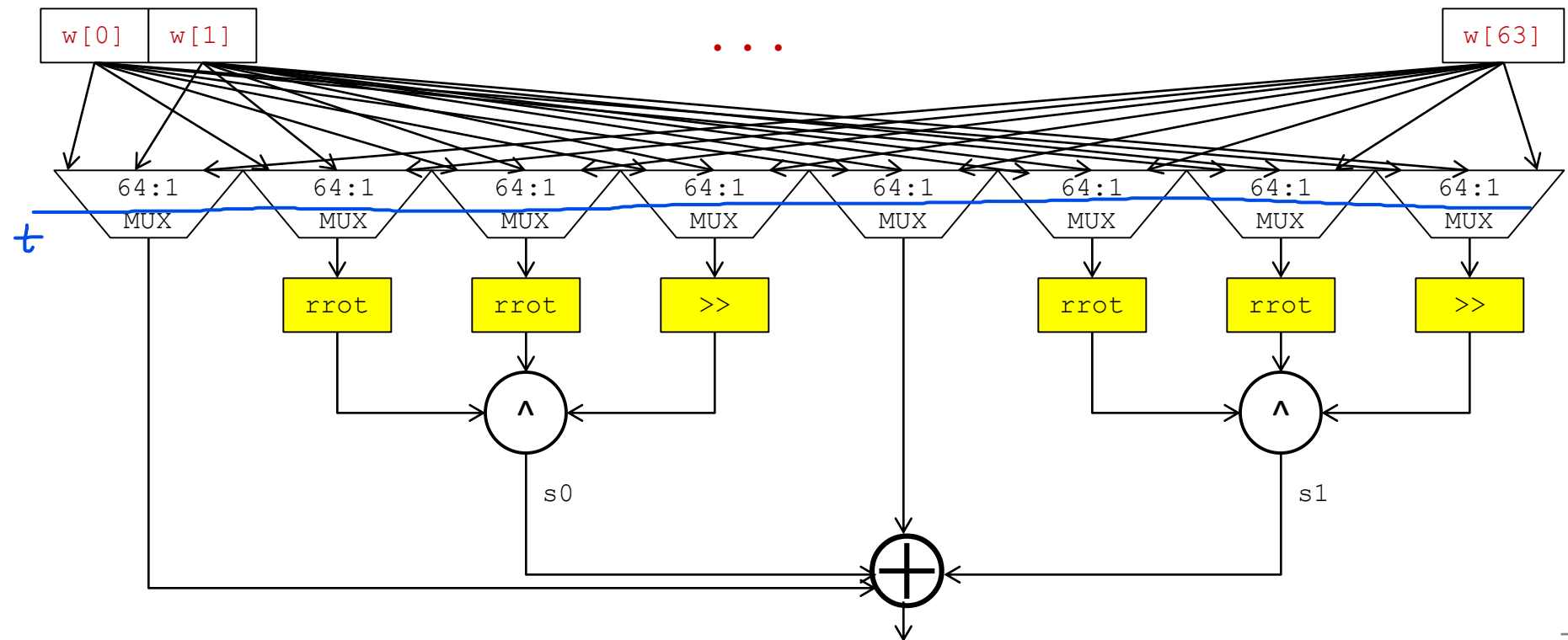
where `rrot` is a function that you can define to implement a circular rotation, but this is very expensive for 2 reasons:

- Need **64** 32-bit registers
- Need expensive **64:1** multiplexors !!!

Hints for W[n] array

- function logic [31:0] wtnew; // function with no inputs
logic [31:0] s0, s1;

```
s0 = rrot(w[t-15],7)^rrot(w[t-15],18)^(w[t-15]>>3);  
s1 = rrot(w[t-2],17)^rrot(w[t-2],19)^(w[t-2]>>10);  
wtnew = w[t-16] + s0 + w[t-7] + s1;  
endfunction
```



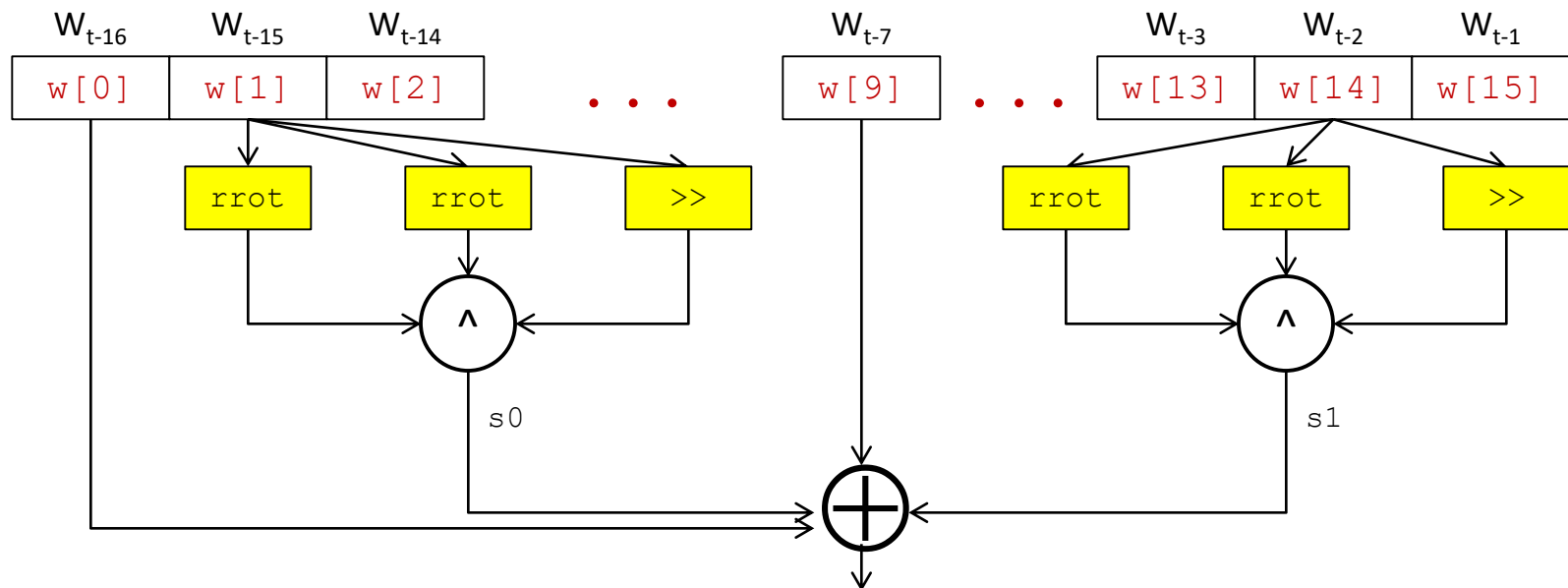
Hints for $W[n]$ array

- We can do the following (i.e, “t-15” is “i = MAX – 15 = 1” for MAX = 16, so therefore W_{t-15} would be $w[1]$). Then

```
function logic [31:0] wtnew; // function with no inputs
    logic [31:0] s0, s1;
```

```
    s0 = rrot(w[1],7)^rrot(w[1],18)^(w[1]>>3);
    s1 = rrot(w[14],17)^rrot(w[14],19)^(w[14]>>10);
    wtnew = w[0] + s0 + w[9] + s1;
```

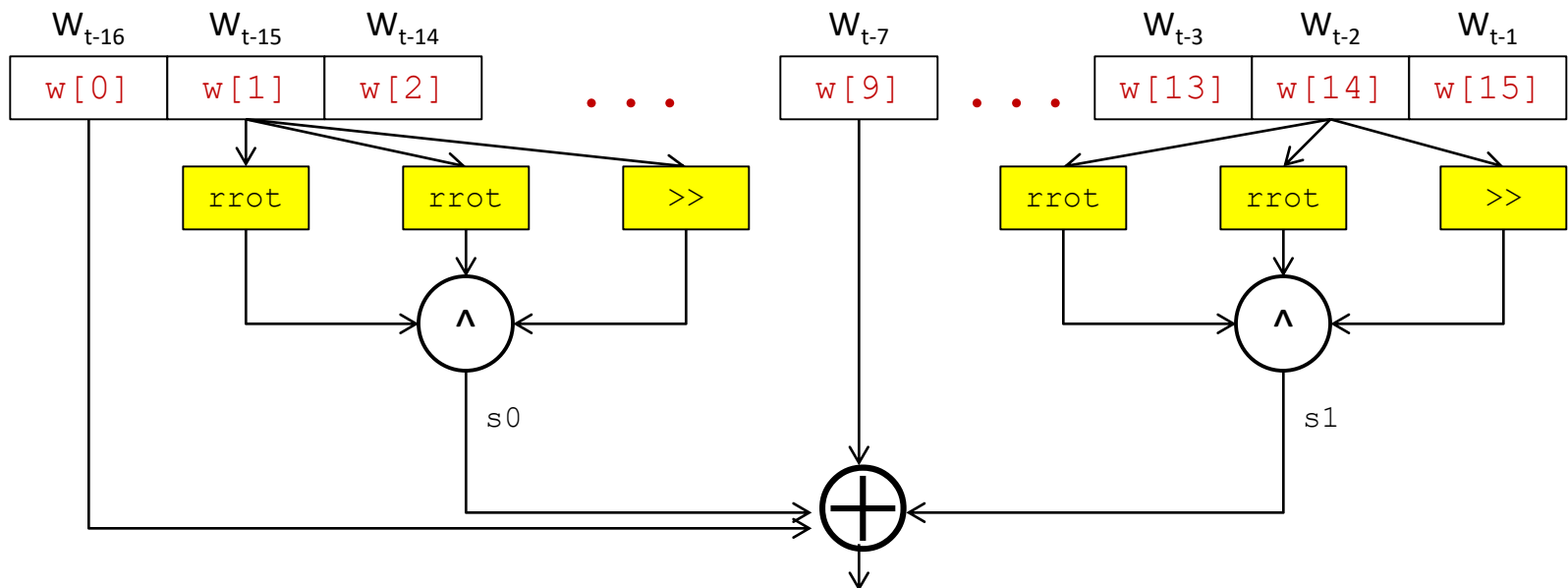
```
endfunction
```



Hints for $W[n]$ array

- Can just write

```
for (int n = 0; n < 15; n++) w[n] <= w[n+1]; // just wires  
w[15] <= wtnew();
```




Possible Results

- A reasonable “median” target:
 - #ALUTs = 1768, #Registers = 1209, Area = 2977
 - Fmax = 107.97 MHz, #Cycles = 147
 - Delay (microsecs) = 1.361, Area*Delay (millesec*area) = 4.053
- With pre-computation of wt:
 - #ALUTs = 1140, #Registers = 1109, Area = 2249
 - Fmax = 155.23 MHz, #Cycles = 149
 - Delay (microsecs) = 0.960, Area*Delay (millesec*area) = 2.159
- Possible to achieve faster Fmax if we pre-compute other parts of the SHA256 logic (more aggressive pipelining)
- Possible to achieve smaller Area*Delay as well

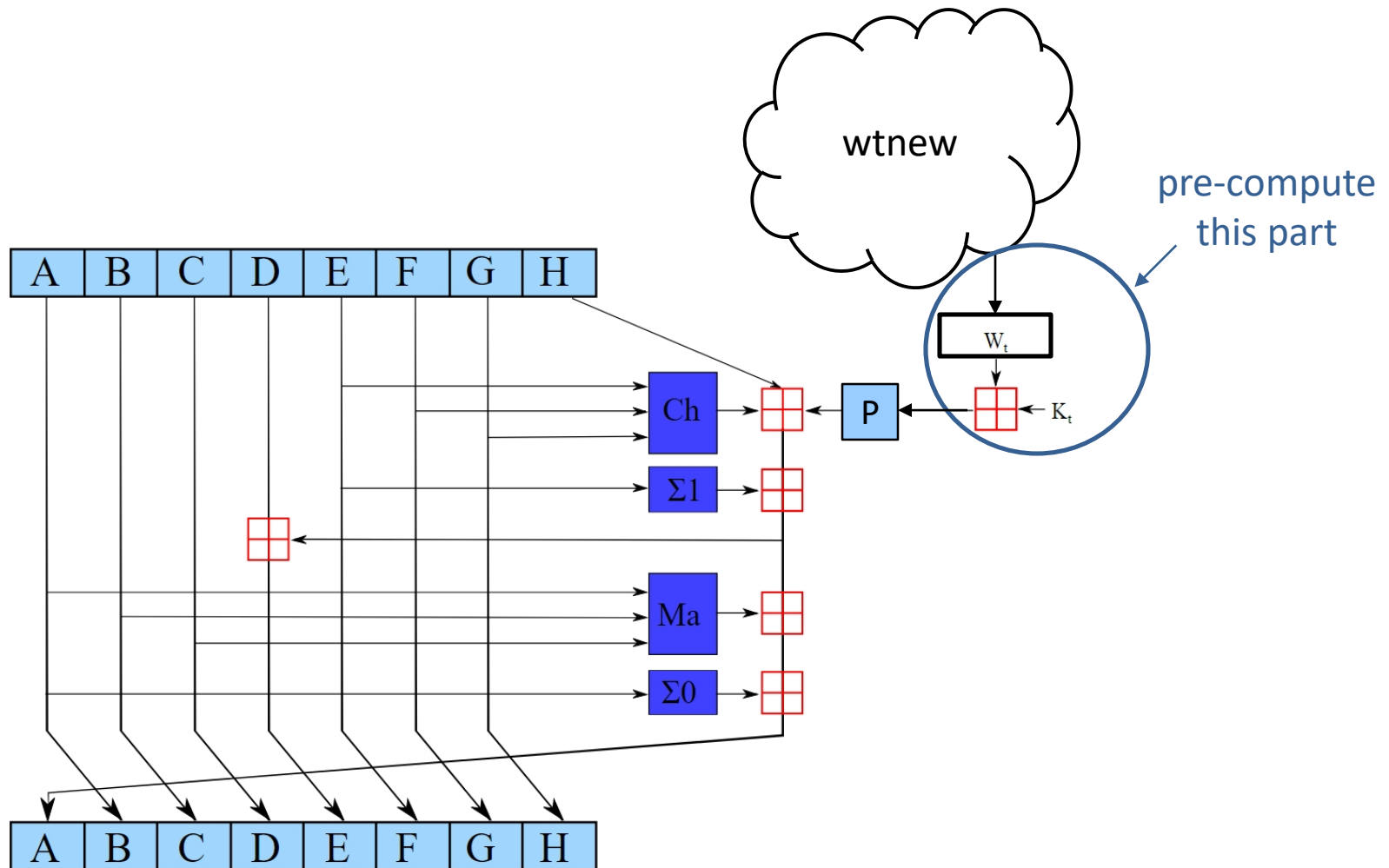
More Aggressive Pipelining

```
function logic [255:0] sha256_op(input logic [31:0] a, b, c, d, e, f, g, h, w, k);
    logic [31:0] S1, S0, ch, maj, t1, t2; // internal signals
begin
    S1 = rightrotate(e, 6) ^ rightrotate(e, 11) ^ rightrotate(e, 25);
    ch = (e & f) ^ ((~e) & g);
    t1 = ch + S1 + h + k + w;
    S0 = rightrotate(a, 2) ^ rightrotate(a, 13) ^ rightrotate(a, 22);
    maj = (a & b) ^ (a & c) ^ (b & c);
    t2 = maj + S0;
    sha256_op = {t1 + t2, a, b, c, d + t1, e, f, g};
end
endfunction
```



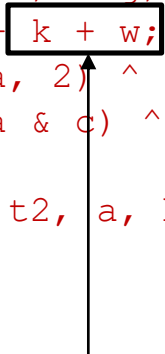
- In general, hard to pipeline this logic because next “a = t1 + t2” is dependent on itself: i.e., t2 = maj + S0, maj = (a & b) ..., S0 = rightrotate(a, 2) ...
- Also hard because next “e = d + t1” is dependent on itself: i.e., t1 = ch + S1, ch = (e & f) ..., S1 = rightrotate(e, 6) ...

Critical Path



More Aggressive Pipelining

```
function logic [255:0] sha256_op(input logic [31:0] a, b, c, d, e, f, g, h, w, k);  
    logic [31:0] S1, S0, ch, maj, t1, t2; // internal signals  
begin  
    S1 = rightrotate(e, 6) ^ rightrotate(e, 11) ^ rightrotate(e, 25);  
    ch = (e & f) ^ ((~e) & g);  
    t1 = ch + S1 + h + k + w;  
    S0 = rightrotate(a, 2) ^ rightrotate(a, 13) ^ rightrotate(a, 22);  
    maj = (a & b) ^ (a & c) ^ (b & c);  
    t2 = maj + S0;  
    sha256_op = {t1 + t2, a, b, c, d + t1, e, f, g};  
end  
endfunction
```



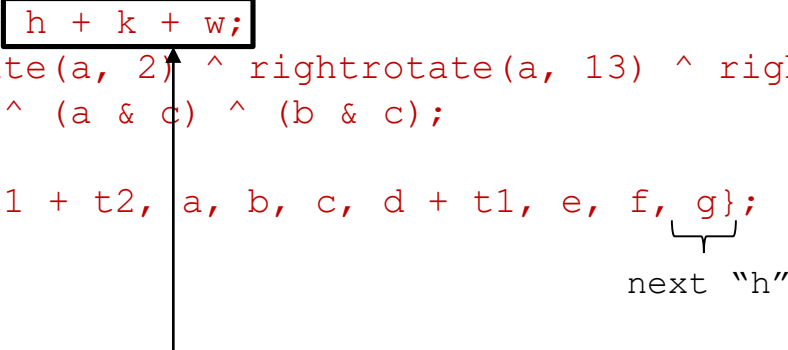
"k" and "w" are not dependent on a, b, c, d, e, f, g, h

Therefore, they can be computed one cycle ahead, but you then have to compute "w" **2 cycles ahead** and use k[t+1] in the pre-computation.

You will need to figure out for yourself how to implement this in SystemVerilog.

More Aggressive Pipelining

```
function logic [255:0] sha256_op(input logic [31:0] a, b, c, d, e, f, g, h, w, k);
    logic [31:0] S1, S0, ch, maj, t1, t2; // internal signals
begin
    S1 = rightrotate(e, 6) ^ rightrotate(e, 11) ^ rightrotate(e, 25);
    ch = (e & f) ^ ((~e) & g);
    t1 = ch + S1 + h + k + w;
    S0 = rightrotate(a, 2) ^ rightrotate(a, 13) ^ rightrotate(a, 22);
    maj = (a & b) ^ (a & c) ^ (b & c);
    t2 = maj + S0;
    sha256_op = {t1 + t2, a, b, c, d + t1, e, f, g};
end
endfunction
```



We can be more aggressive. Next "h" is equal to "g", but "h" is not dependent on itself.

Hint: need "h" one cycle ahead.

You will need to figure out for yourself how to implement this in SystemVerilog.