



# ESL anyone?

**September 28, 2010**

**Chad Spackman**

**RTP site design manager**

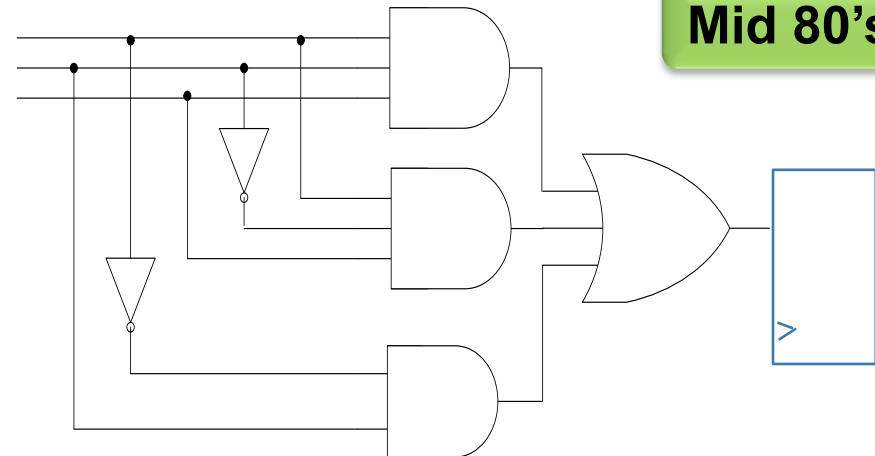
# Agenda

- ▶ No Agenda -> I promise
  
- ▶ Discussion about:
  - The origins of RTL, what's good and what isn't
  - Motivation behind a more abstract design entry flow
  - ESL Observed difficulties
  - Examples of some things to pay attention to in ESL flows
  - Future... Replace RTL?

## Why was RTL (verilog vhdl) invented? What problem was being solved? Answer: It's hard to describe decision trees with gates

```
If (address == my_addr)
{
  if (receiver_enabled && ! queue_full)
    write_enable = 1;
  else
    drop_packet = 1;
} else if (address == `MULTICAST)
...

```



**Operators:** + - \* / << are handy but nonessential. Doing such things schematically, or using Boolean equations, is not hampered by the means of design entry

**Flops and Muxes :** A means to describe flipflops, muxing, etc are provided. **They are painful**

**Wiring:** Instantiation and wiring are included. **They are included for completeness and can be very painful**

**The task of designing digital circuits did not really change much when schematics gave way to HDL**

- Designers break the overall function into data plane & control plane
- They coax the language into producing what they can draw, and in many cases, have drawn
- There is time spent on sensitivity lists, the proper use of case/casex, etc.
- The technology advance was in taking if-else trees and converting them to Boolean equations
- Readability and self documentation took a back seat.

# Why the shift towards C/C++? What problem was being solved? Answer: State Extraction

late '90s  
to early  
2000's

```
always @(posedge clk)
begin
  if (reset == `RESET_LEVEL)
  begin
    send_state <= `IDLE;
  end
  else
  begin
    case (send_state)
    `IDLE:
    begin
      if (send_header)
        send_state <= `H1;
      else
        send_state <= `IDLE;
    end
    `H1:
    begin
      send_state <= `H2;
    end
  end
end
```

Control Plane

```
always @(posedge clk)
begin
  if (reset == `RESET_LEVEL)
  begin
    data_to_send <= 64'h0;
    num_bits <= 8'd0;
  end
  else
  begin
    if (send_state == `H1)
    begin
      data_to_send <= 64'h0;
      num_bits <= 8'd0;
    end
    if (send_state == `H2)
    begin
      data_to_send <= {`TSTMP, `FLG, `CM, `ID2, `ID1};
      num_bits <= 8'd64;
    end
  end
end
```

Data Plane

## Procedural (a 1/2 step to C)

```
forever
begin
  data_to_send <= 64'h0;
  num_bits <= 8'd0;
  `CLOCK

  while (!send_header)
  `CLOCK

  data_to_send <= {`TSTMP, `FLG, `CM, `ID2, `ID1};
  num_bits <= 8'd64;

  `CLOCK
  data_to_send <= {`RFC51_H, `OS, `XFL};
  more stuff
  .
  .
  `CLOCK
end
```

- The step from Boolean to RTL was tiny and did not change a designers thought process
- The step from RTL to procedural is a significant change to design methodology
- Full support of the C language places extreme emphasis on coding style to get desirable outcome

## Why has the adoption of ESL been so slow?

Answer: there a big difference from previous design entry innovations

### ▶ RTL pros and cons

- Cryptic
- Difficult to read if poorly commented
- Operates at the gate level
- High degree of control
- New embodiments start from scratch
- Re-use has made up for deficiencies

### ▶ C/C++ pros and cons

- Obvious
- Self documenting and highly readable
- Operates quite far above the gate level
- Less control ?
- New embodiments may start from existing code
- ESL requires a change in how we think about design



**Biggest Complaint about raised abstraction -> “I can’t see the design”**  
**The reason: We may not start from scratch**

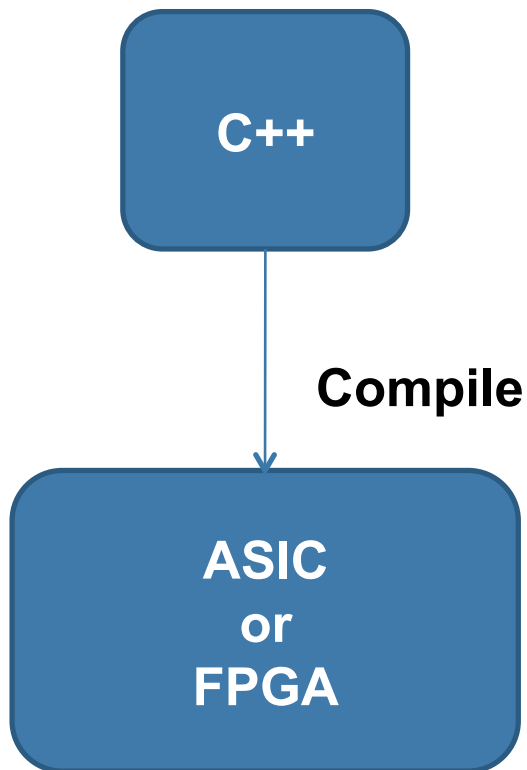
## Challenges to ESL and C/C++ adoption

- ▶ C/C++ is a “big” language (a challenge to learn)
  - Every line of C has a hardware embodiment
  - Every hardware embodiment has a line of C
- ▶ ESL is a bigger step than previous design entry innovations
  - Designers must stop thinking about state machines
  - Designers must stop thinking about wires and synchronization flags between blocks of parallelism
- ▶ Since C/C++ existed for another purpose, there is existing code
  - With Verilog or VHDL the designer is forced to start from scratch. Every task is oriented towards making efficient hardware.
  - If the C code already exists, it lends itself to being manually rendered compliant and then tossed to the compiler to “see how it does”
  - This is a guaranteed path to failure yet lead to extremely rapid exploration

## Things that make adoption easier

- ▶ The designer needs to know what of the language is supported
  - A list of exclusions simply makes life confusing
  - Full support of C or the C subset of C++ is a reasonable expectation
  - SystemC class extensions and libraries can only help to standardize methodologies
  
- ▶ Education
  - Obviously, but common sense rules
  - C is obvious
  - We can come to unanimous and unambiguous agreement of what a piece of C code produces
  - Think data path
  - Forget about state!!!!

## C/C++ – Reasonable expectations



-C/C++ extends the concept of state inference

- data types
- function calls
- global variables
- pointer support
- Clock insertion

Nice to have

← This one has ramifications

- No coding style should cause C semantics to come unglued

Myth:

- C is sequential and cannot describe hardware
- In fact sequential coding is the point
- C is a highly descriptive HDL
- The hardware produced is obvious

## What cannot be expected

- ▶ Compile any industry code and get “good” hardware
  - Companies love to put this sort of thing in sales brochures
- ▶ A compiler cannot be expected to introduce system level parallelism.
  - This is an architectural task for now just as it is in the software domain
- ▶ A compiler cannot detect and remove unintended datapath (examples on this)
- ▶ A compiler cannot remove long paths created by coding of unnecessary dependencies (same as Verilog/VHDL)



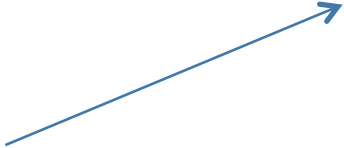
## Some Pitfalls

```

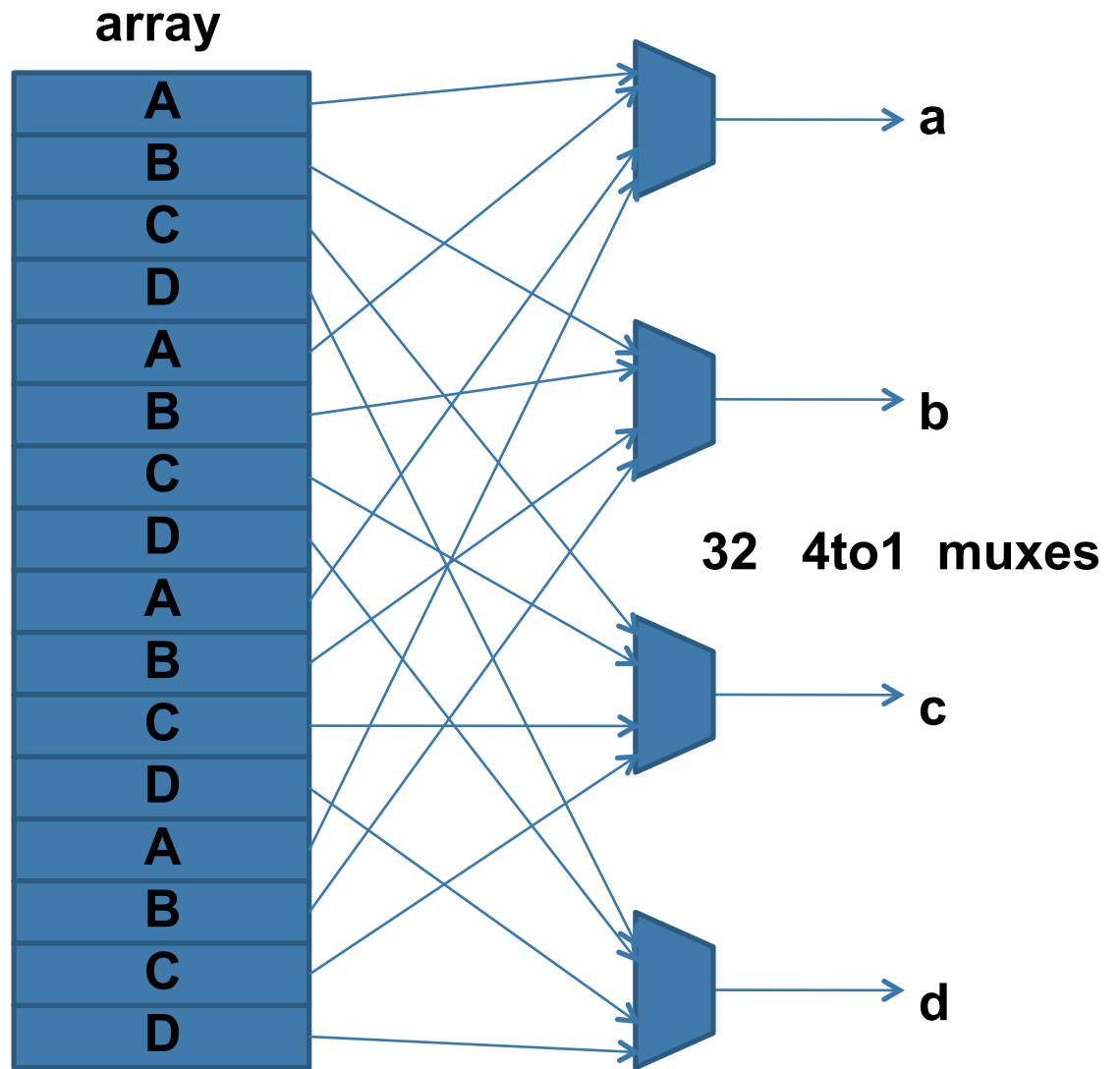
uint8_t    a, b, c, d;
uint8_t    array[16];

for (i = 0; i < 16; i = i + 4)
{
    a = array[i];
    b = array[i + 1];
    c = array[i + 2];
    d = array[i + 3];
}

```

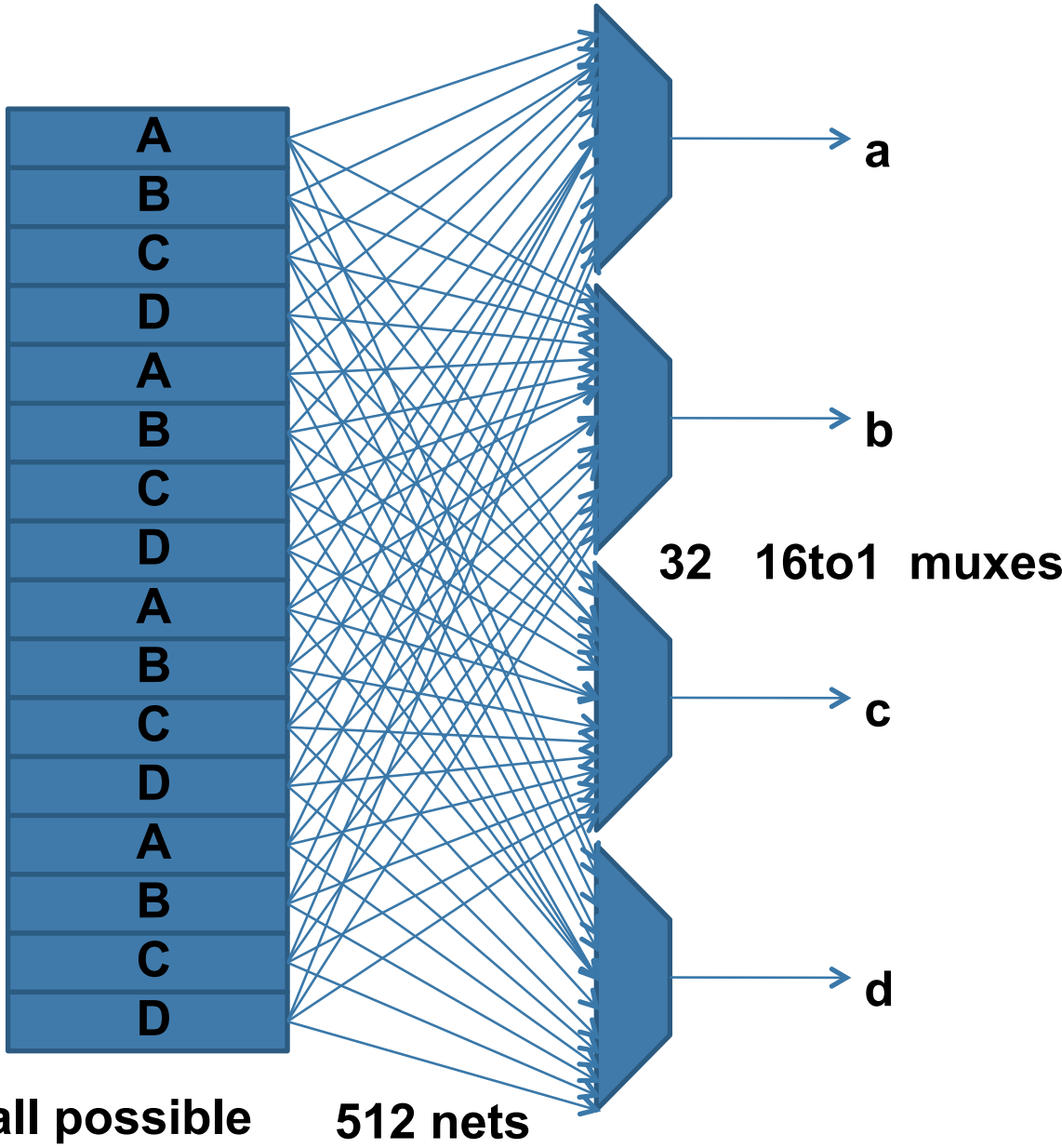


**Intended architecture**



```
uint8_t    a, b, c, d;
uint8_t    array[16];

for (i = 0; i < 16; i = i + 4)
{
    a = array[i];
    b = array[i + 1];
    c = array[i + 2];
    d = array[i + 3];
}
```



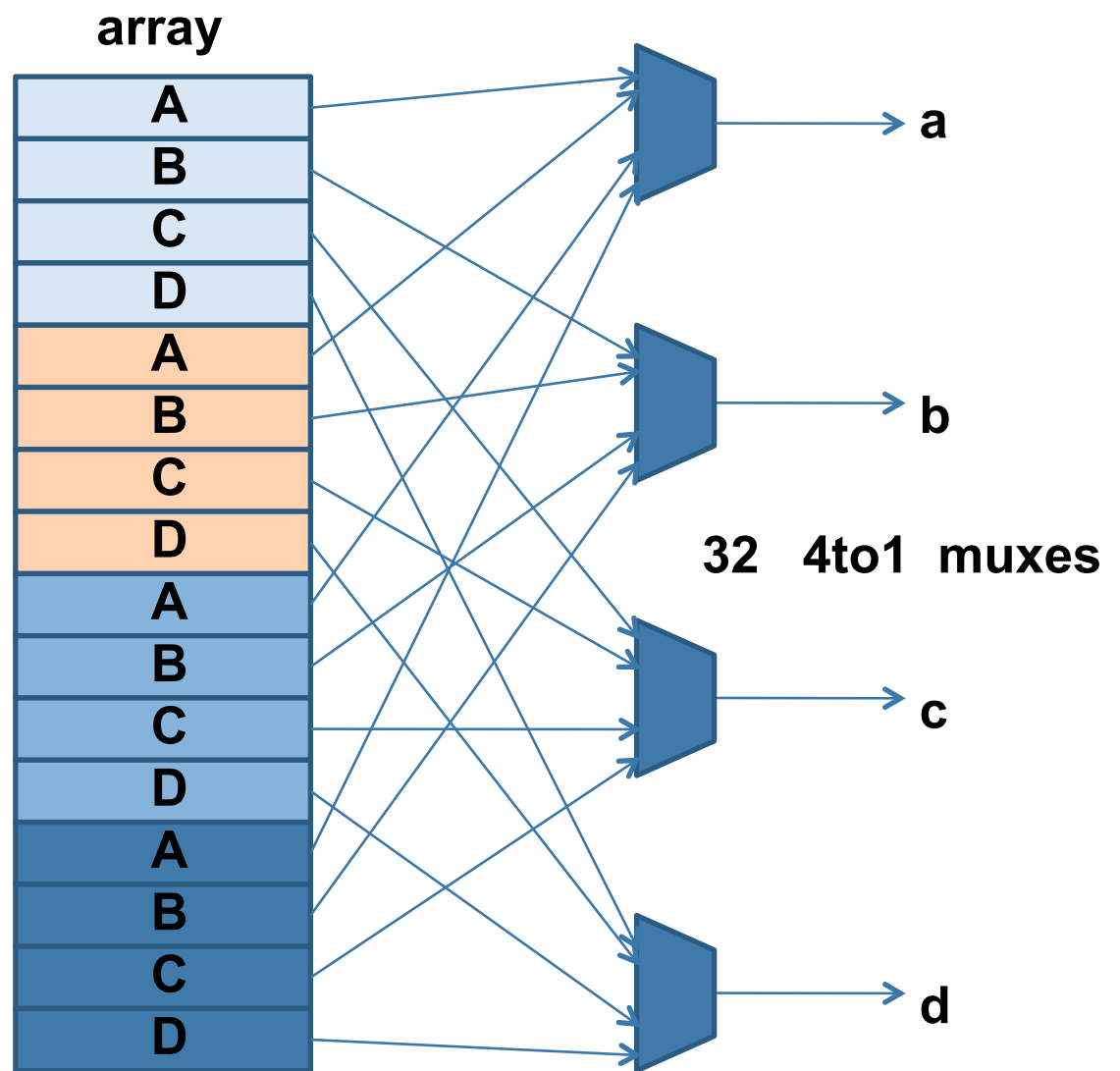
**Whoops**  
The compiler accounts for all possible values of 'i'

## Alternative coding

```
uint8_t    a, b, c, d;
struct fred
{
    uint8_t a;
    uint8_t b;
    uint8_t c;
    uint8_t d;
}

struct fred array[4];

for (i = 0; i < 4; i = i + 1)
{
    a = array[i].a;
    b = array[i].b;
    c = array[i].c;
    d = array[i].d;
}
```



# SHA256 as released from <http://b-con.us>

```
a = ctx_state[0];
b = ctx_state[1];
c = ctx_state[2];
d = ctx_state[3];
e = ctx_state[4];
f = ctx_state[5];
g = ctx_state[6];
h = ctx_state[7];
```

## 1) Grab Data

```
for (i = 0; i < 8; ++i)
{
    uint32_t tmp3;
```

## 2) Expand it into first 16 of M[64]

```
    m[2 * i] = SWAP ((uint32_t) (ctx_data[i]));
    tmp3 = ctx_data[i] >> 32;
    m[2 * i + 1] = SWAP (tmp3);
}
```

## 3) Do sha

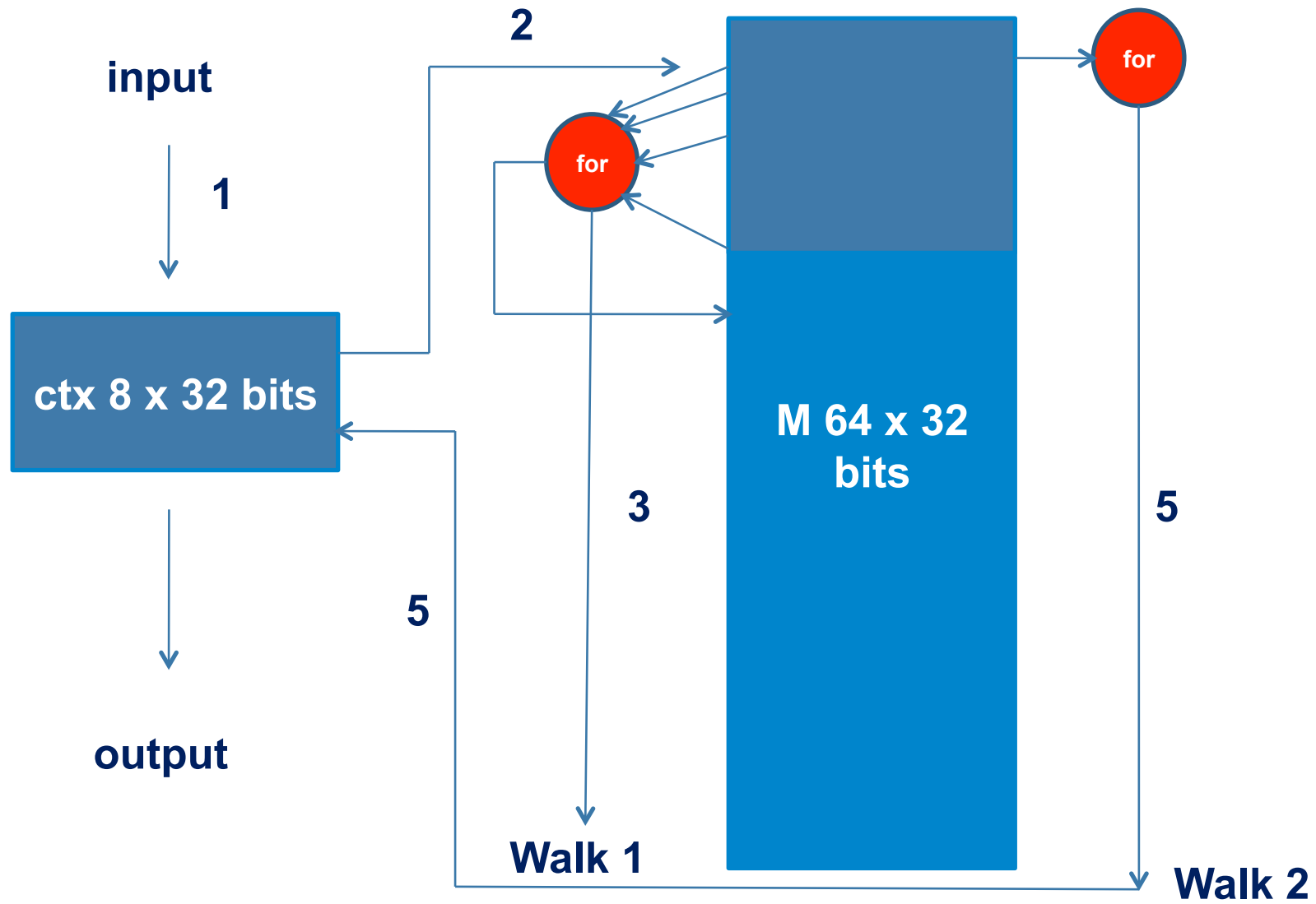
```
ctx_state[0] += a;
ctx_state[1] += b;
ctx_state[2] += c;
ctx_state[3] += d;
ctx_state[4] += e;
ctx_state[5] += f;
ctx_state[6] += g;
ctx_state[7] += h;
```

## 4) return Data

```
for (i = 16 ; i < 64; ++i)
    m[i] = SIG1(m[i-2]) + m[i-7] + SIG0(m[i-15]) + m[i-16];
```

```
for (i = 0; i < 64; ++i) {
    t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i];
    t2 = EP0(a) + MAJ(a,b,c);
    h = g;
    g = f;
    f = e;
    e = d + t1;
    d = c;
    c = b;
    b = a;
    a = t1 + t2;
}
```

# Innate architecture of the algorithm



# Deriving and architecture

- Is storage usage something to be concerned about?
  - No, 64 x 32 is the largest and that isn't large
- So where might gates come from?
  - Look at a simplified version of the first 'for' loop

```
for (i = 16 ; i < 64; ++i)
    m[i] = m[i-2] + m[i-7] + m[i-15] + m[i-16];
```

- This loop will walk every entry on the flipflop based array
  - It will place 4 reading muxes ON EVERY M[] FLIPFLOP
  - It will place 1 writing mux on ON EVERY M[] FLIPFLOP
  - A boat load of gates and a boat load of wire
- Look at a simplified version of the second 'for' loop

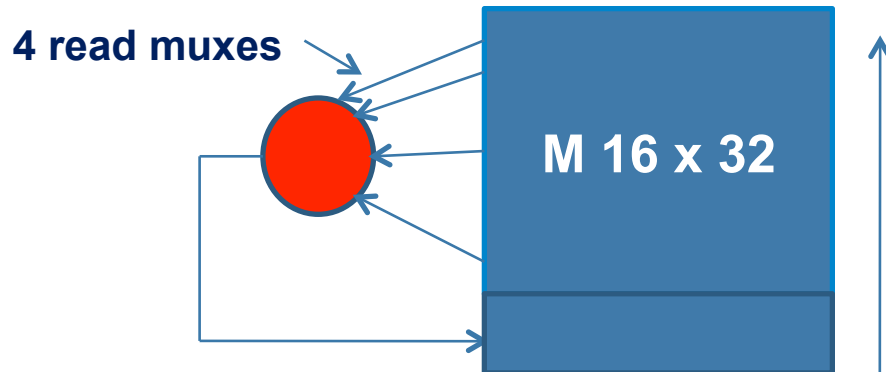
```
for (i = 0; i < 64; ++i) {
    t1 = k[i] + m[i];
```

  - This loop walks the array again, top to bottom
  - It doesn't begin until after the first loop finishes
  - It discards each linear value of m[i] after its use
- These are our clues to a better architecture
  - We never need more than 16 in the history of the first loop
  - We don't need to wait to start the second loop

**Sorry – even with  
ESL you have to  
“think” about  
architecture!**

## An alternate architecture

With our knowledge of the linear sweeps the loops make, and the limited history required, lets make the data travel to the algorithm rather than the taking the algorithm to the data



Roll the data through and toss it as it falls off the end

**This becomes**

```
for (i = 16 ; i < 64; ++i)
    m[i] = m[i-2] + m[i-7] + m[i-15] + m[i-16];

for (i = 0; i < 64; ++i) {
    t1 = k[i] + m[i];
```

**This**

```
for (i = 0 ; i < 64; ++i)
{
    m[16] = m[14] + m[9] + m[1] + m[0];
    t1 = k[i] + m[0];

    unroll;
    for (j = 0; j < 16; ++j) // roll the data through and discard
        m[j] = m[j + 1];
}
```

Glue the muxes down

# The final code

```
a = ctx_state[0];
b = ctx_state[1];
c = ctx_state[2];
d = ctx_state[3];
e = ctx_state[4];
f = ctx_state[5];
g = ctx_state[6];
h = ctx_state[7];

unroll;
for (i = 0; i < 8; ++i)
{
    uint32_t tmp3;

    m[2 * i] = SWAP ((uint32_t) (ctx_data[i]));
    tmp3 = ctx_data[i] >> 32;
    m[2 * i + 1] = SWAP (tmp3);
}

wait();

ctx_state[0] += a;
ctx_state[1] += b;
ctx_state[2] += c;
ctx_state[3] += d;
ctx_state[4] += e;
ctx_state[5] += f;
ctx_state[6] += g;
ctx_state[7] += h;
```

```
for (i = 0 ; i < 64; ++i)
{
    m[16] = SIG1(m[14]) + m[9] + SIG0(m[1]) + m[0]; // Next m[15] to shift in

    t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[0];
    t2 = EP0(a) + MAJ(a,b,c);
    h = g;
    g = f;
    f = e;
    e = d + t1;
    d = c;
    c = b;
    b = a;
    a = t1 + t2;

    unroll; // shift pipe
    for (j = 0; j < 16; ++j)
        m[j] = m[j + 1];
}
```

**We expressed a drastically different architecture by changing 4 lines of code  
We obeyed the rules of native simulation.**

## The future of ESL?

- ▶ The industry has compensated for some of the difficulties of RTL
  - Design reuse
  - The new world of verification
- ▶ The choice to use ESL must be “need” driven from the bottom up
  - Designers choice
  - Not CTO’s choice
- ▶ Results must be as predictable as RTL
  - Responsibility lies with the ESL vendor to:
    - EDUCATE
    - DELIVER
  - Verification methods must be part of it



Thank you!