EE4218 L5 - HLS

1 Context

Suppose we want to implement hardware for the code below.

```
int a[16], b[16], c[16], d[16], e[16], f[16], x[16], y[16];
for(int i=0; i<16; i++)
{
    x[i] = a[i]*b[i]+c[i];
    y[i] = (d[i]+e[i])*f[i];
}
```

Assume . . .

- *a[i]*'s to *f[i]*'s are seperate memories that can be read asynchronously
- *x[i]*'s and *y[i]*'s are seperate memories that are written synchronously
- Adder has delay of 5ns, Multiplier has delay of 10ns
- Circuit is resource-dominated (ie. Area and performance depend on a few well-characterized functional resources)

2 Single-cycle vs Multi-cycle design

2.1 Single Cycle

Single cycle design is where all of the logic is done *combinationally* in one cycle

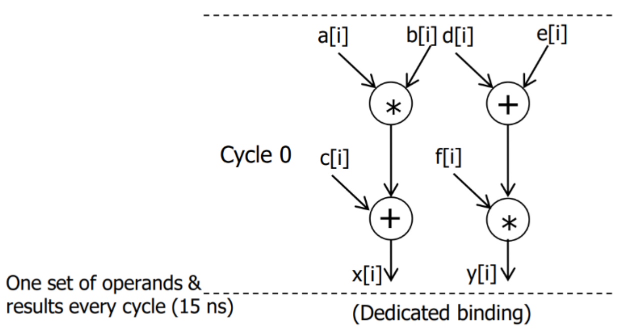
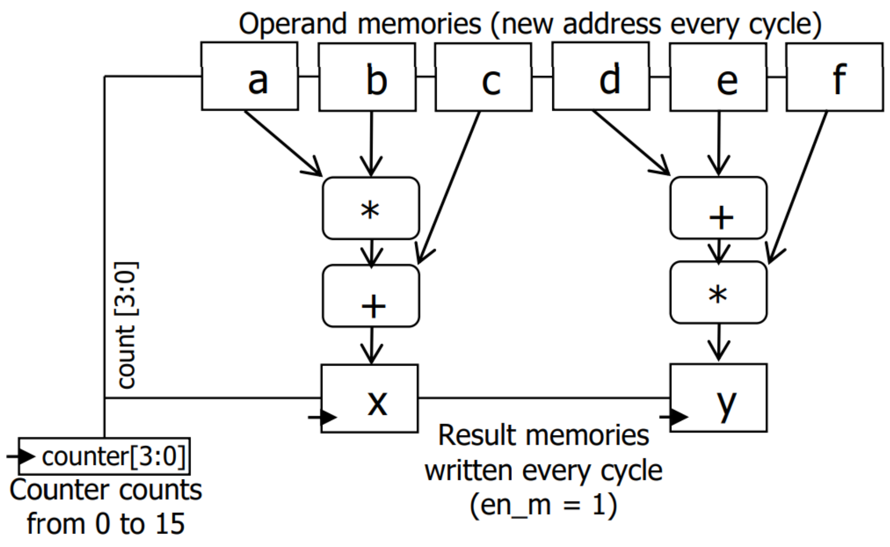


Figure 1: Single Cycle - Scheduling

Notice that scheduling is trivial, since all computations are done combinationally. Therfore, there is only a single time-step in the schedule.

Furthur note that no resource-sharing is possible, since all operations are done in the same cycle

- 2 ALUs, 2 MULs
- Period = 15ns
- Latency (per iteration) = 1 cycle
- Trip Count (number of loop iterations) = 16
- Total latency = 16 cycles



Operand memories (new address every cycle)
a
b
c
d
e
f
count [3:0]
x
y
counter[3:0]
Counter counts from 0 to 15
Result memories written every cycle (en_m = 1)

Figure 2: Single Cycle - Hardware implementation
Note that *counter* is an abstraction of our control logic

2.2 Multi Cycle (default in older version of HLS)

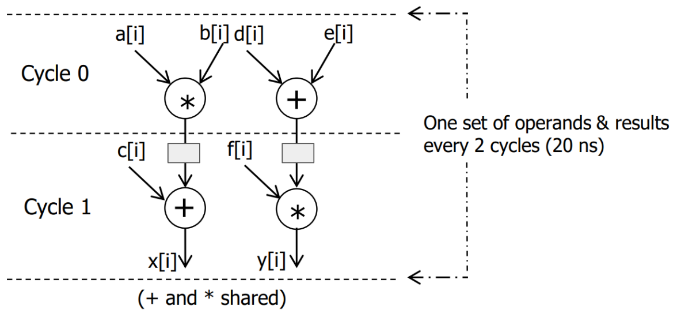
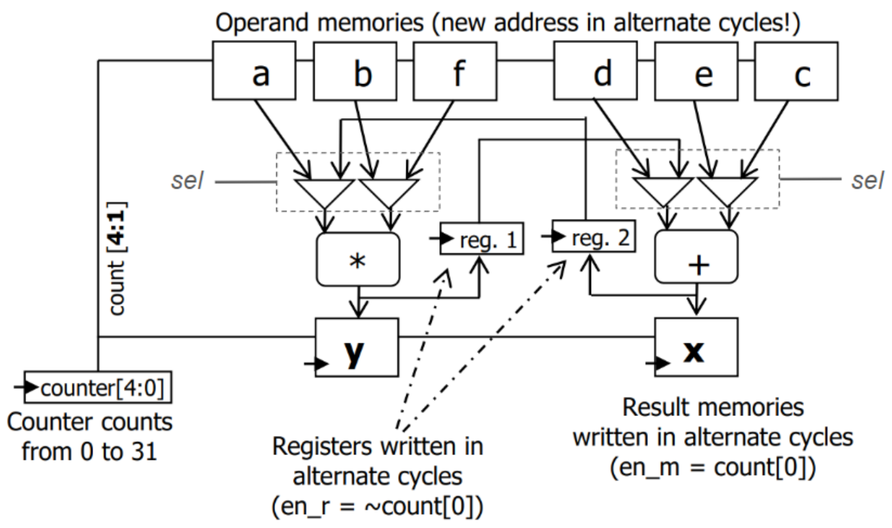


Figure 3: Multi Cycle - Scheduling

Note the inclusion of temp registers to store the intermediate results (with MUX *sel* signals channeling it appropriately)

Resource-sharing is now possible, and the clock period (bounded by the slower MUL unit) is decreased as well

- 1 ALU, 1 MULs
- Period = 10ns
- Latency (per iteration) = 2 cycle
- Trip Count (number of loop iterations) = 16
- Total latency = 32 cycles



Operand memories (new address in alternate cycles!)
a
b
f
d
e
c
sel
sel
count [4:1]
reg. 1
reg. 2
y
x
counter[4:0]
Counter counts from 0 to 31
Registers written in alternate cycles (en_r = ~count[0])
Result memories written in alternate cycles (en_m = count[0])

Figure 4: Multi Cycle - Hardware implementation

3 Pipelining (default in newer version of HLS)

In our multi-cycle design before, we had to complete the computation of the i^{th} result before we could proceed to compute the $(i + 1)^{th}$ result

Pipelining issues a new set of inputs every j^{th} cycle instead, which we define as *Initiation Interval*(II)

Therefore, the **absolute** latency per datapoint becomes less revelant, we are more interested in *throughput* (datapoints processed per *unit time*)

3.1 Pipelined (*II = 1*)

Issuing a new set of inputs every cycle

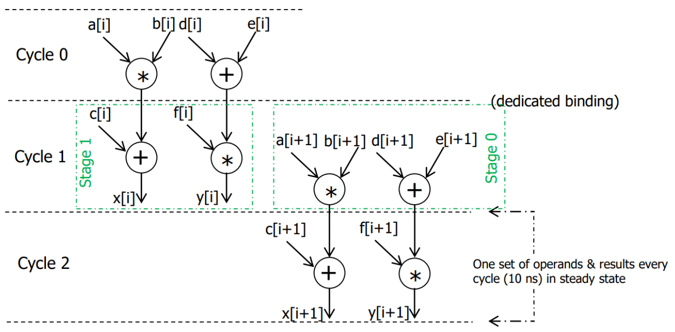


Figure 5: Pipeline (II=1) - Scheduling

We implicitly assume the presence of temporary regs (to store data from i^{th} to $(i^{th} + 1)$ cycle)

There is overlap in execution of the previous and current set of operands, and at *steady-state* we complete a computation at the end of every cycle

- 2 ALUs, 2 MULs
- Period = 10ns
- Latency (per iteration) = 2 cycles
- Trip Count (number of loop iterations) = 16
- Total latency = (*II* *tripCount) + 1 cycle overhead to fill pipeline = 17 cycles

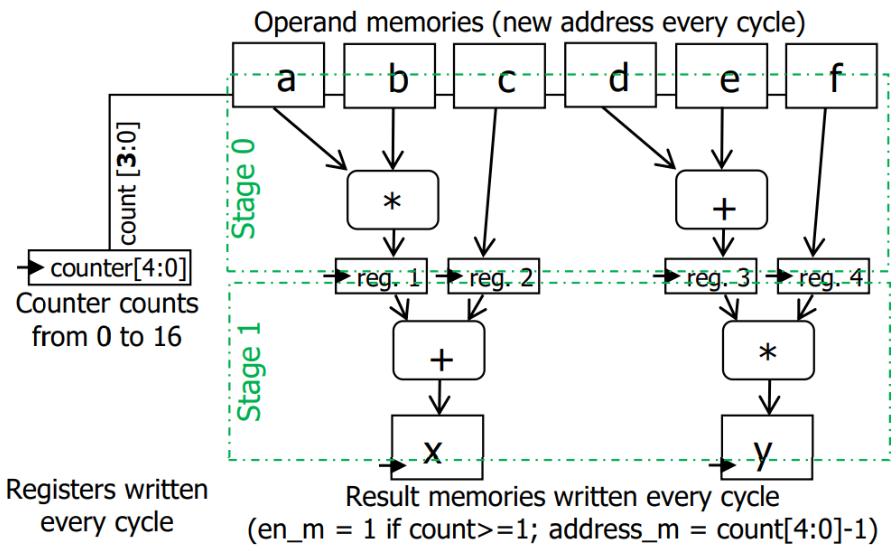

Operand memories (new address every cycle)
a
b
c
d
e
f
count [3:0]
counter[4:0]
Counter counts from 0 to 16
Stage 0
*
+
reg. 1
reg. 2
reg. 3
reg. 4
Stage 1
+
*
x
y
Registers written every cycle
Result memories written every cycle
(en_m = 1 if count>=1; address_m = count[4:0]-1)

Figure 6: Pipeline (II=1) - Hardware implementation

3.2 Pipelined (*II = 2*)

Issuing a new set of inputs every other cycle

- In *II*=1 scenario, each combinational operation was fit into one clock period, with the **slowest** combinational operation determining the fastest clock period.
- However, what if we could 'stretch' the **slowest** combinational operation over multiple cycles? Then we could furthur decrease the clock period.
- Assume that we are looking at 'stretching' the MUL operation, since it is the slowest. There are two ways to accomplish this ...
  1. Modify the **combinational** MUL unit to a **sequential** one, that will now take 2 clock cycles to complete
  2. Keep the MUL unit as combinational, but manipulate the *write-enable* of the registers that capture it's results. This requires a *set_multicycle_path* constraint if we were designing it manually (more in L6 - Timing Analysis)
     - 2.1. Combinational logic is not instantaneous in reality, there is $t_{propagation-delay}$ which denotes the maximal delay of the combinational logic.
     - 2.2. Think of it as $t_{propagation-delay} > t_{clock-period}$, therefore the write enable of the capturing register must be manipulated accordingly

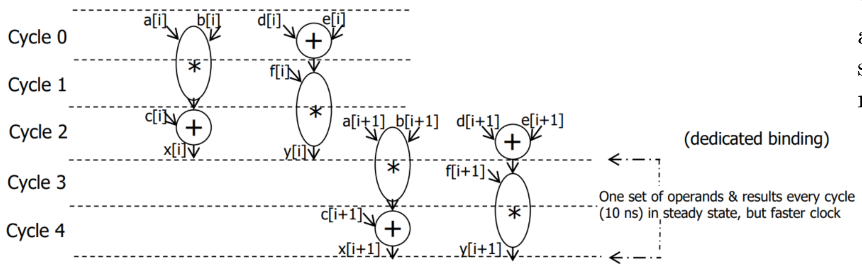


Figure 7: Pipeline (II=2) - Scheduling

With *II*=2, our clock period is 5ns and we issue a new set of inputs every 2 cycles (10ns). This is similar to *II*=1, but our clock is operating quicker now.

- 2 ALUs, 2 MULs
- Period = 5ns
- Latency (per iteration) = 3 cycles
- Trip Count (number of loop iterations) = 16
- Total latency = (*II* *tripCount) + 2 cycle overhead to fill pipeline = 34 cycles

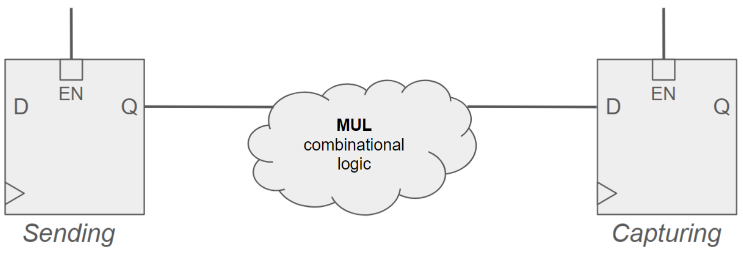


Figure 8: *EN* of the capturing register is asserted only on alternate clock cycles

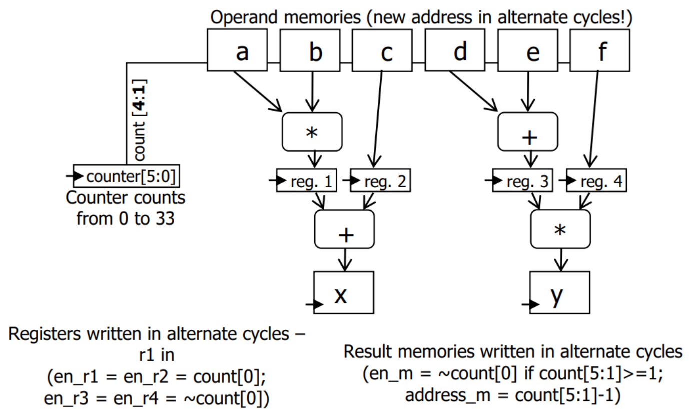

Operand memories (new address in alternate cycles!)
a
b
c
d
e
f
count [4:1]
counter[5:0]
Counter counts from 0 to 33
*
+
reg. 1
reg. 2
reg. 3
reg. 4
+
*
x
y
Registers written in alternate cycles – r1 in (en_r1 = en_r2 = count[0]; en_r3 = en_r4 = ~count[0])
Result memories written in alternate cycles (en_m = ~count[0] if count[5:1]>=1; address_m = count[5:1]-1)

Figure 9: Pipeline (II=2) - Hardware implementation

4 More aggressive optimizations

In all the previous sections, per cycle we have at most one new input. Is it possible to give more than one new input per cycle?

By applying some behavioral level optimizations (eg. *loop unrolling*), we can.

4.1 Loop Unrolling

Loop unrolling requires memories that are capable of reading multiple values at a time.

In the example below for example, we would require *a[i]* and *a[i+1]* to be stored in seperate memories, so that they can be accessed in the same clock cycle. One method to do so would be *cyclical array partitioning* (ie. memory interleaving) by a factor of 2.

```
int a[16], b[16], c[16], d[16],
         e[16], f[16], x[16], y[16];
for(int i=0; i<16; i+=2)
{
    x[i]  = a[i]*b[i]+c[i];
    y[i]  = (d[i]+e[i])*f[i];
    x[i+1]  = a[i+1]*b[i+1]+c[i+1];
    y[i+1]  = (d[i+1]+e[i+1])*f[i+1];
}
return 0;
```

Figure 10: Loop unrolling by a factor of 2

4.1.1 Unrolling + Partition + Pipelined

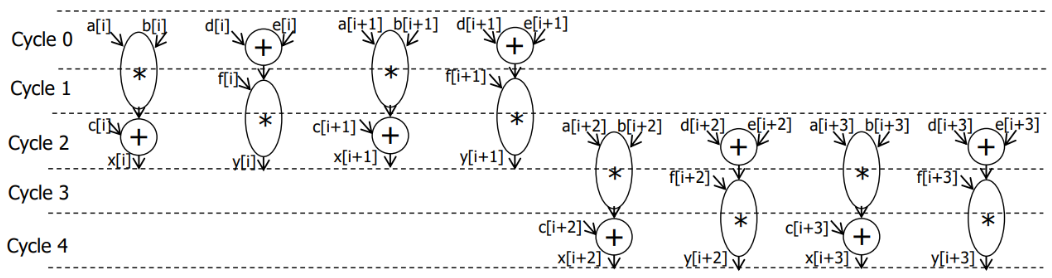


Figure 11: Combining all optimizations together

- Combining loop unrolling (by a factor of 2) and pipelining, we can give **two** sets of inputs every other cycle
- Note that resources will be duplicated, thus we need 4 ALUs, 4 MULs now
- Period = 5ns, Latency = 3 cycles, Trip count = 8 cycles
- Total latency = (II *tripCount) + 2 cycle overhead to fill pipeline = 18 cycles