CSE565M: Acceleration of Algorithms in Reconfigurable Logic

Learn by Doing: Finite Impulse Response (FIR) Filters (Pt. 2)

Anthony Cabrera

FL24::L04

Washington University in St. Louis

Table of contents

1. Calculating Performance
2. Operation Chaining
3. Code Hoisting
4. Loop Fission
5. Loop Unrolling
6. Loop Pipelining

Calculating Performance

Apples to Apples Comparisons

Need to make sure we compare two systems using the same metric

- How *fast* it runs
  - Latency? (how long one iteration takes)
  - Throughput? (rate at which you can process input)
  - operates at $X \frac{bits}{second}$
  - performs $Y \frac{ops}{second}$
- Other perf measures
  - $\frac{filter\ \ operations}{second}$
  - how many multiply-accumulates per second: $\frac{MACs}{s}$

Operation Chaining

Operation Chaining

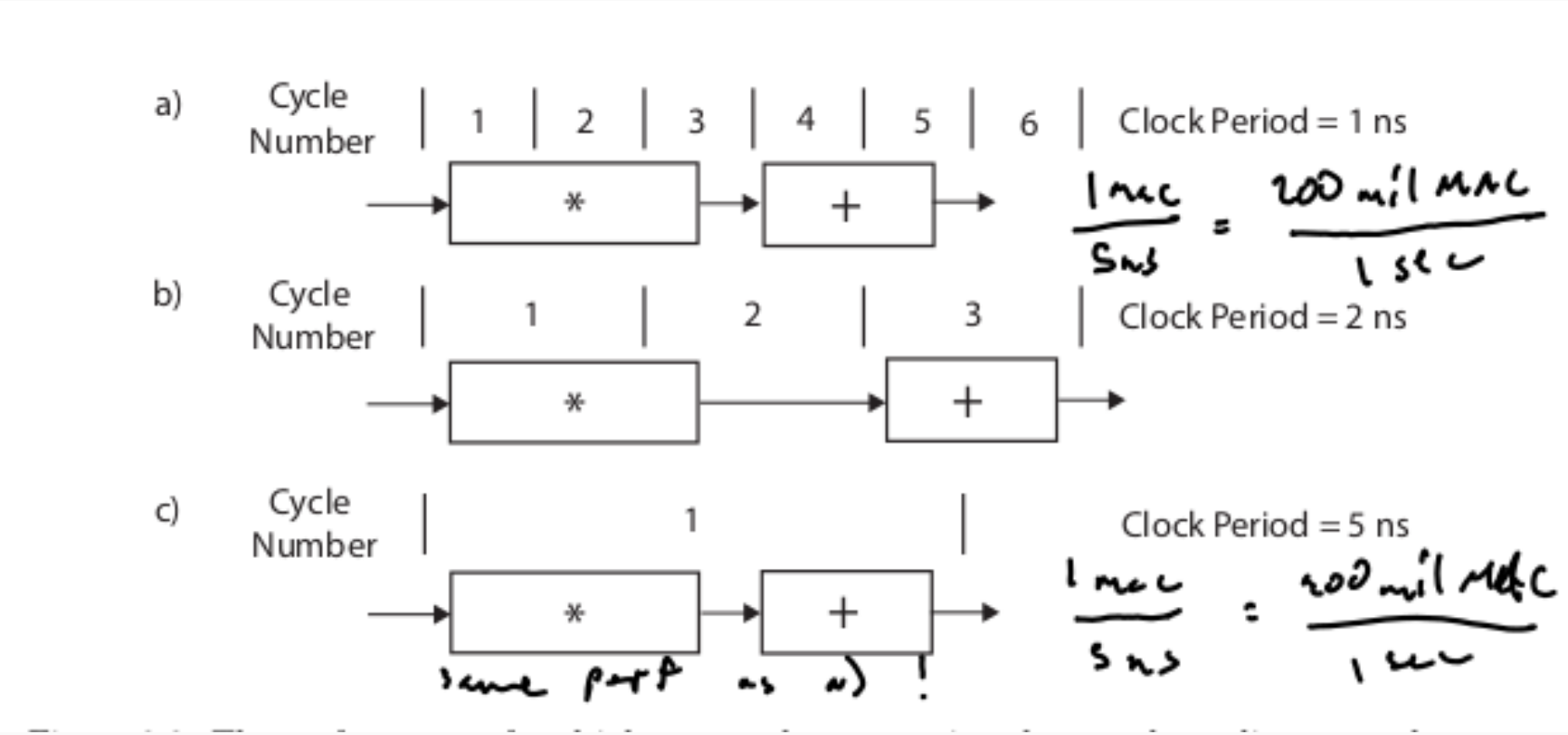

Figure 1: Effects of the hardware compiler applying operation chaining at different clock frequencies.

Code Hoisting

Base FIR Architecture

```
1  #define N 11
2  #include "ap_int.h"
3
4  typedef int coef_t;
5  typedef int data_t;
6  typedef int acc_t;
7
8  void fir(data_t *y, data_t x) {
9    coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
10   static data_t shift_reg[N];
11   acc_t acc;
12   int i;
13
14   acc = 0;
15 Shift_Accum_Loop:
16   for (i = N - 1; i >= 0; i--) {
17     if (i == 0) {
18       acc += x * c[0];
19       shift_reg[0] = x;
20     } else {
21       shift_reg[i] = shift_reg[i - 1];
22       acc += shift_reg[i] * c[i];
23     }
24   }
25   *y = acc;
26 }
```

Figure 2: A functionally correct, but highly unoptimized, implementation of an 11 tap FIR filter.

Code Hoisting in the FIR Example

The `if`/`else` logic is inefficient. Why?

- For every control structure in the code, the tool creates logical hardware that checks if the condition is met, which is executed in every iteration of the loop.
- Furthermore, this conditional structure limits the execution of the statements in either the `if` or `else` branches; these statements can only be executed after the `if` condition statement is resolved, i.e., pipelining becomes inefficient!

Code Hoisting in the FIR Example

The `if/else` logic is inefficient. Why?

- For every control structure in the code, the tool creates logical hardware that checks if the condition is met, which is executed in every iteration of the loop.
- Furthermore, this conditional structure limits the execution of the statements in either the `if` or `else` branches; these statements can only be executed after the `if` condition statement is resolved, i.e., pipelining becomes inefficient!

How to Fix with Code Hoisting?

- The `if` statement checks when `i == 0`, which happens only on the last iteration.
- Therefore, the statements within the `if` branch can be "hoisted" out of the loop.
- That is we can execute these statements after the loop ends, and then remove the `if/else` control flow in the loop.
- Finally, we must change the loop bounds from executing the "0th" iteration. This transform is shown in Figure 3. This shows just the changes that are required to the `for` loop.

How to Fix with Code Hoisting?

- The `if` statement checks when `i == 0`, which happens only on the last iteration.
- Therefore, the statements within the `if` branch can be "hoisted" out of the loop.
- That is we can execute these statements after the loop ends, and then remove the `if/else` control flow in the loop.
- Finally, we must change the loop bounds from executing the "0th" iteration. This transform is shown in Figure 3. This shows just the changes that are required to the `for` loop.

How to Fix with Code Hoisting?

- The `if` statement checks when `i == 0`, which happens only on the last iteration.
- Therefore, the statements within the `if` branch can be "hoisted" out of the loop.
- That is we can execute these statements after the loop ends, and then remove the `if/else` control flow in the loop.
- Finally, we must change the loop bounds from executing the "0th" iteration. This transform is shown in Figure 3. This shows just the changes that are required to the `for` loop.

How to Fix with Code Hoisting?

- The `if` statement checks when `i == 0`, which happens only on the last iteration.
- Therefore, the statements within the `if` branch can be "hoisted" out of the loop.
- That is we can execute these statements after the loop ends, and then remove the `if/else` control flow in the loop.
- Finally, we must change the loop bounds from executing the "0th" iteration. This transform is shown in Figure 3. This shows just the changes that are required to the `for` loop.

Result after Code Hoisting

```
1 Shift_Accum_Loop:
2 for (i = N - 1; i > 0; i--) {
3   shift_reg[i] = shift_reg[i - 1];
4   acc += shift_reg[i] * c[i];
5 }
6
7 acc += x * c[0];
8 shift_reg[0] = x;
```

Figure 3: Removing the conditional statement from the `for` loop creates a more efficient hardware implementation.

The end results is a much more compact implementation that is ripe for further loop optimizations, e.g., unrolling and pipelining.

NOTE

Even targeting a traditional CPU, this optimization removes conditional logic that just makes the code a little more reasonable and will generate less instructions.

Loop Fission

Loop Fission

- Loop fission takes the two tasks in the single loop and decomposes them into their own loop.
  - this allows us to perform optimizations separately on each loop
  - this can be advantageous especially in cases when the resulting optimizations on the split loops are different
  - performing loop fission isn't necessarily always better though

```
1  TDL:
2  for (i = N - 1; i > 0; i--) {
3    shift_reg[i] = shift_reg[i - 1];
4  }
5  shift_reg[0] = x;
6
7  acc = 0;
8  MAC:
9  for (i = N - 1; i >= 0; i--) {
10   acc += shift_reg[i] * c[i];
11 }
```

Loop Fission

- Loop fission takes the two tasks in the single loop and decomposes them into their own loop.
  - this allows us to perform optimizations separately on each loop
  - this can be advantageous especially in cases when the resulting optimizations on the split loops are different
  - performing loop fission isn't necessarily always better though

```
1  TDL:
2  for (i = N - 1; i > 0; i--) {
3    shift_reg[i] = shift_reg[i - 1];
4  }
5  shift_reg[0] = x;
6
7  acc = 0;
8  MAC:
9  for (i = N - 1; i >= 0; i--) {
10   acc += shift_reg[i] * c[i];
11 }
```

Loop Fission

- Loop fission takes the two tasks in the single loop and decomposes them into their own loop.
  - this allows us to perform optimizations separately on each loop
  - this can be advantageous especially in cases when the resulting optimizations on the split loops are different
  - performing loop fission isn't necessarily always better though

```
1  TDL:
2  for (i = N - 1; i > 0; i--) {
3    shift_reg[i] = shift_reg[i - 1];
4  }
5  shift_reg[0] = x;
6
7  acc = 0;
8  MAC:
9  for (i = N - 1; i >= 0; i--) {
10   acc += shift_reg[i] * c[i];
11 }
```

Loop Unrolling

Loop Unrolling Part 1

- replicates the body of the loop and reduces the number of iterations by the same factor
- in the best case, iterations are independent and can substantially increase the available parallelism
  - note the loose ends because the divisor does not divide cleanly
  - also note the decrement reflecting the factor of 2
  - also also note that this is how we get those different implementations of the FIR filter from Ch. 1 w.r.t. how many tap coefficients we have access to during one iteration of the shift register loop

```
1 TDL:
2 for (i = N - 1; i > 1; i = i - 2) {
3   shift_reg[i] = shift_reg[i - 1];
4   shift_reg[i - 1] = shift_reg[i - 2];
5 }
6 if (i == 1) {
7   shift_reg[1] = shift_reg[0];
8 }
9 shift_reg[0] = x;
```

Auto Unrolling

```
1  static data_t shift_reg[N];
2  #pragma HLS ARRAY_PARTITION variable = shift_reg complete dim = 0
3
4  TDL : for (i = N - 1; i > 1; --i) {
5  #pragma HLS PIPELINE II = 1
6    shift_reg[i] = shift_reg[i - 1];
7  }
8  if (i == 1) {
9    shift_reg[1] = shift_reg[0];
10 }
11 shift_reg[0] = x;
```

Figure 4: Example of using ‘pragma‘s to automatically unroll loop.

Unrolling the MAC Loop

```
1  acc = 0;
2  MAC:
3  for (i = N - 1; i >= 3; i -= 4) {
4    acc += shift_reg[i] * c[i] + shift_reg[i - 1] * c[i - 1] +
5           shift_reg[i - 2] * c[i - 2] + shift_reg[i - 3] * c[i - 3];
6  }
7
8  for (; i >= 0; i--) {
9    acc += shift_reg[i] * c[i];
10 }
```

Figure 5: Manually unrolling the MAC loop.

- Don't forget the loose ends!

Loop Unrolling Part 2

- note that just because you unroll by a big factor doesn't necessarily mean it's possible
  - e.g., mapping the shift register to a BRAM would be limited by the number of read and write ports
- also unroll the computaiton loop (fig 2.6)
  - don't forget the loose ends after you rewrite!

Loop Pipelining

Loop Pipelining Pt 1

By default, Vitis HLS will synthezie ‘for‘ loops in a **sequential** manner

- e.g., the ‘for‘ loop in Figure 2.1 will perform each iteration of the loop after the other.
- that is, all of the statements in the second iteration happen only wnen all of the statements from the first iteration are complete and so on

We can do better! But we as the programmer have to help the Vitis HLS tool (using **loop pipelining**)

- consider the previous MAC ‘for‘ loop. the operations are
  - read `c[]`
  - read `shift_reg[]`
  - *: multiple the values from `c[]` and `shift_reg[]`
  - +: accumulate this mulitplied result into the `acc` variable

Loop Pipelining Pt 2

- a corresponding schedule for the above is shown in Fig 2.7
- each read takes 2 cycles
- cycle 1: provde address to memory
- cycle 2: read memory using the address
- each read can be done in parallel
- * can occur in cycle 2
- assume it takes 3 cycles, i.e., finished after cycle 4
- the '+' operation is *operation chained* to start and complete during cycle 4
- thus, the entire body of the MAC 'for' loop takes 4 cycles to complete

Pipeline Schedule

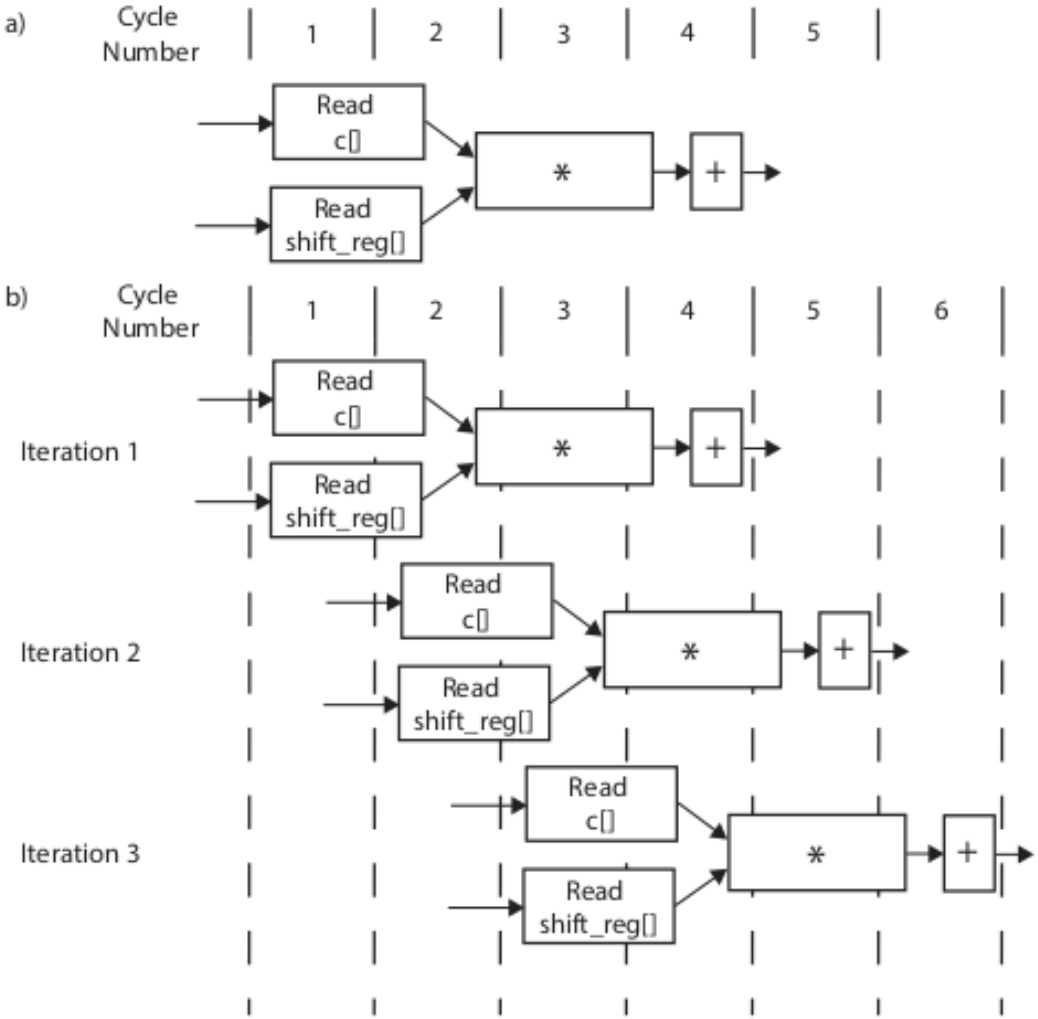

Figure 2.7: Part a) shows a schedule for the body of the MAC **for** loop. Part b) shows the schedule for three iterations of a pipelined version of the MAC **for** loop.

Loop Pipelining Pt 3

- performance metrics associated with ‘for‘ loop:
- iteration latency: number of cycles it takes to perform one iteration of the loop body
- iteration latency of the above example is 4 cycles
- ‘for‘ loop latency:
- number of cycles required to complete the entire execution of the loop
- includes time to calcualte the initialization statement (e.g., ‘i = 0‘), and the increment statment
- assuming that these header statements can be done in parallel with the loop body execution, the LS tool reports the latency of this MAC ‘for‘ loop as 44 cycles
- number of iterations multipled by the iteration latency

Loop Pipelining Pt 4

- Loop pipelining is an optimization that overlaps multiple iterations of a ‘for‘ loop
- Overlap iteration execution
- total loop latency of 14 vs 44!
- Loop initiation interval (II) is another important metric
- defined as the number of iterations until the next iteration of the loop can start
- in the example case, II=1 so a new iteration can start every cycle
- II can be explicitly set using a directive
- ‘pragma HLS pipeline II=2‘ informs HLS tool to attempt ‘II=2‘.
- any ‘for‘ loop can be pipelined, so we consider the TDL ‘for‘ loop
- ‘pragma HLS pipeline II=1‘

References i