A Self-Timed Radix-2 FFT Design

Mertcan Temel
mert@utexas.edu

February 6, 2018
Introduction

• A design methodology/State machine examples using the Link-joint model (introduced in 2015 by Roncken et al.)
  • Unsigned Multiplier
  • Signed Multiplier
  • Complex Multiplier
  • Radix-2 FFT

• Everything implemented as circuit generators in DE (but no proofs just yet)
Outline

• Link-Joint Model
• Asynchronous Register
• Multipliers
  • Unsigned
  • Signed
  • Complex
• Radix-2 Decimation-in-time FFT Summary
• Self-timed Radix-2 FFT Module
• Summary and Future Work
**Link-Joint Model**

### Links
- Storage elements
- D-latches for Data
- SR latch for full status: is data valid?
- `wr_fire`: write new data & mark as full
- `rd_fire`: read data & mark as empty

### Joint
- Data Processing Elements
- Implements fire rules: decides when data should proceed
- Goes between two links
- `wr_fire`: write to links
- `rd_fire`: read from links
We can have as many joints and links connected together *as long as they alternate*!
Link-Joint Model (cntd.)

**Complex Link**
- A module/encapsulation that starts and ends with links.
- Can be connected to joints on each side.
- Denoted by a rectangle with rounded corners

**Joint to Link Arrow:**
- Full (link->joint)
- Din (joint->link)
- wr_fire (join->link)

**Link to Joint Arrow:**
- Full (link->joint)
- Dout (link->joint)
- rd_fire (join->link)
Link-Joint Model Restrictions

Restriction 1:
A link cannot have multiple writer/reader joints

Restriction 2:
A link cannot be read and written by the same joint

How do you intervene and load data to modules?

How do you update the same data when processing?
Asynchronous Register

Here, we propose an asynchronous register module.

- Load initial data to Li
- Write data to Lin (during processing)
- Read data from Lout (during processing)
- Denote the module with double lined rectangles

**Function of Ji:**

```java
if (full(Li))
    Lout = Li;
    fire Li, Lin, Lout;
elseif (full(Lin) && empty(Lout))
    Lout = Lin;
    fire Lin, Lout;
end if;
```

- Now, we can load initial data, and use the same joint to update data
- We can **write** and **still read the previous data**
Asynchronous n-bit Unsigned Multiplier

- Registers are initialized by some external joint writing to the module.
- L-busy indicates the module is not ready (prefull) for new input.
- Registers, M.cand, and L-busy all use the same wr_fire signal that originates externally.
- The module behaves like a link (complex link), only it has two full signals.
- Double arrow indicates that the joint both reads and writes.
Asynchronous n-bit Unsigned Multiplier (cntd.)

- **J-mul** performs shift and add to multiply.
  - if MSB(Multiplier) is 1, add M.cand to Acc
  - Shift Multiplier and Acc
  - When Counter=n, clear everything and buffer Acc to Result.

- **M.cand** is just a link as we only read from it.

- The module can **pipeline** requests. Can start another calculation before Result is cleared.
1. An external joint pre-fires to start
2. Ji1, Ji2, Ji3 initialize successor links
   • Lcnt1=0
   • Lacc1=0
   • L3=L2
3. Center Joint J1 processes and propagates data
   • If MSB(L3) then
     _lshf and add L1
   else
     _lshf and buffer
4. Repeat when cnt<31
5. J1 finishes if cnt==31.
   • Wr. res to L5
   • Release all prelinks
Asynchronous 32-bit Unsigned Multiplier

1. An external joint pre-fires to start
2. Ji1, Ji2, Ji3 initialize successor links
   • Lcnt1=0
   • Lacc1=0
   • L3=L2
3. Center Joint J1 processes and propagates data
   • If MSB(L3) then 
     lshf and add L1
   else
     lshf and buffer
4. Repeat when cnt<31
5. J1 finishes if cnt==31.
   • Wr. res to L5
   • Release all prelinks
1. An external joint pre-fires to start
2. Ji1, Ji2, Ji3 initialize successor links
   - Lcnt1=0
   - Lacc1=0
   - L3=L2
3. Center Joint J1 processes and propagates data
   - If MSB(L3) then lshf and add L1
     else
       lshf and buffer
4. Repeat when cnt<31
5. J1 finishes if cnt==31.
   - Wr. res to L5
   - Release all prelinks
Asynchronous 32-bit Unsigned Multiplier

1. An external joint pre-fires to start
2. Ji1, Ji2, Ji3 initialize successor links
   - Lcnt1=0
   - Lacc1=0
   - L3=L2
3. Center Joint J1 processes and propagates data
   - If MSB(L3) then _lshf and add L1
   - else _lshf and buffer
4. Repeat when cnt<31
5. J1 finishes if cnt==31.
   - Wr. res to L5
   - Release all prelinks
1. An external joint pre-fires to start
2. Ji1, Ji2, Ji3 initialize successor links
   - Lcnt1=0
   - Lacc1=0
   - L3=L2
3. Center Joint J1 processes and propagates data
   - If MSB(L3) then lshf and add L1
   - else lshf and buffer
4. Repeat when cnt<31
5. J1 finishes if cnt==31.
   - Wr. res to L5
   - Release all prelinks
1. An external joint pre-fires to start
2. Ji1, Ji2, Ji3 initialize successor links
   - Lcnt1=0
   - Lacc1=0
   - L3=L2
3. Center Joint J1 processes and propagates data
   - If MSB(L3) then
     _lshf and add L1
   else
     _lshf and buffer
4. Repeat when cnt<31
5. J1 finishes if cnt==31.
   - Wr. res to L5
   - Release all prelinks
1. An external joint prefires to start
2. Ji1, Ji2, Ji3 initialize successor links
   - Lcnt1=0
   - Lacc1=0
   - L3=L2
3. Center Joint J1 processes and propagates data
   - If MSB(L3) then _lshf and add L1
   - else _lshf and buffer
4. Repeat when cnt<31
5. J1 finishes if cnt==31.
   - Wr. res to L5
   - Release all prelinks
1. An external joint pre-fires to start
2. Ji1, Ji2, Ji3 initialize successor links
   • Lcnt1=0
   • Lacc1=0
   • L3=L2
3. Center Joint J1 processes and propagates data
   • If MSB(L3) then lshf and add L1
   else lshf and buffer
4. Repeat when cnt<31
5. J1 finishes if cnt==31.
   • Wr. res to L5
   • Release all prelinks
1. An external joint pre-fires to start
2. Ji1, Ji2, Ji3 initialize successor links
   - Lcnt1=0
   - Lacc1=0
   - L3=L2
3. Center Joint J1 processes and propagates data
   - If MSB(L3) then
     - Lshf and add L1
   - else
     - Lshf and buffer
4. Repeat when cnt<31
5. J1 finishes if cnt==31.
   - Wr. res to L5
   - Release all prelinks
1. An external joint prefires to start
2. Ji1, Ji2, Ji3 initialize successor links
   - Lcnt1=0
   - Lacc1=0
   - L3=L2
3. Center Joint J1 processes and propagates data
   - If MSB(L3) then lshf and add L1
   - else lshf and buffer
4. Repeat when cnt<31
5. J1 finishes if cnt==31.
   - Wr. res to L5
   - Release all prelinks
1. An external joint pre-fires to start
2. Ji1, Ji2, Ji3 initialize successor links
   • Lcnt1=0
   • Lacc1=0
   • L3=L2
3. Center Joint J1 processes and propagates data
   • If MSB(L3) then _lshf and add L1
   else _lshf and buffer
4. Repeat when cnt<31
5. J1 finishes if cnt==31.
   • Wr. res to L5
   • Release all prelinks

Asynchronous 32-bit Unsigned Multiplier
1. An external joint pre-fires to start
2. Ji1, Ji2, Ji3 initialize successor links
   - Lcnt1=0
   - Lacc1=0
   - L3=L2
3. Center Joint J1 processes and propagates data
   - If MSB(L3) then lshf and add L1
   - else lshf and buffer
4. Repeat when cnt<31
5. J1 finishes if cnt==31.
   - Wr. res to L5
   - Release all prelinks
Asynchronous n-bit Signed Multiplier

- **Unsigned Multiplier** is used as a link
- A queue is used to make use of **pipelining capability** of Unsigned Multiplier.
- J1 two’s complement inputs if they’re negative
- J2 buffers
- J3 two’s complement the result if the result is to be negative.
- Signed Multiplier can pipeline **4 requests** before Result is cleared
Asynchronous n-bit Complex Multiplier

• **Signed Multiplier** is used as a link.

• A queue is used to make use of **pipelining capability** of Signed Multiplier.

• Performs 4 real multiplication per a complex multiplication.

• It can **pipeline 7 requests** before Result is cleared.

![Diagram of Asynchronous n-bit Complex Multiplier](image-url)
Radix-2 Decimation-in-Time FFT Summary

**Goal**: is to calculate Discrete Fourier Transform (DFT) efficiently

**A recursive algorithm**
- Input divided into **two sets** and **N/2-point DFT** is calculated on each
- **Results** are **paired up** and multiplied by a constant $W_N^m$

**There are log(N) steps** in each of which **N complex multiplications** are performed.

$$W_N^m = e^{-j2\pi m/N}$$
Radix-2 Decimation-in-Time FFT Summary (cntd.)

The iterative solution is needed before implementing

- An formula to redistribute inputs for initialization
- Iterate over each step and then each number
- Find the formula to determine the constant $W_N^m$
- Find the formula to determine the other number $n_2$
- Update numbers with $n_1 \times W_N^m + n_2$

- When initializing:
  $$next_i = inputs_{(reverse-bits \ i)}$$

- When calculating:
  $$\text{if } i>y$$
  $$next_i = prev_y + prev_i \times W_N^z$$
  $$\text{else}$$
  $$next_i = prev_i + prev_y \times W_N^z$$

where

- $i =$ index of number being processed [0 N]
- $s =$ current step [0 logN]
- $y =$ $i$’s $s$th bit flipped
- $z =$ $i$ $\ll$ logN-s

Old values (prev) need to be retained as they’re used twice!
Asynchronous Radix-2 FFT

- For N-point FFT, numbers are stored in **N registers**.

  - **J1**
    - Keeps track of **state**
    - Selects **number pair**
    - Selects what **W** to read
  
  - **J2**
    - Performs addition
    - Writes the resulting number to a register

- **Index Q.** : index of the number processed
- **Number Q.** : pair of the number processed

- **Old values are kept** until next step while **J2** writes on the registers!

\[
k = \text{ceil}(\log(\log n) + \log n)
\]
Summary and Future Work

- An asynchronous “register” module is proposed.
  - Hopefully, it’ll be used developing even more complex machines

- A radix-2 FFT module with a single multiplier implemented as circuit generators in DE

- Plan to work on the functional correctness of the given modules
- Plan to introduce new designs using the link-joint model