浮動小数点演算器をつくった話

X: アンドゥー＠carbon_hero

Github: dokunira

浮動小数点演算器

FloPoCo

オープンソースの浮動小数点演算器ジェネレータ
以下のようなコマンドでVHDLの浮動小数点演算器を生成してくれる便利なツール

```
# flopoco frequency=120 target=Zynq7000 IEEEFPFMA wE=8 wF=23
```

丸めはroundTiesToEvenのみ
指定した周波数より遅いことがある(retimingをonにしても)

FPNew

オープンソースのRISC-Vに準拠した浮動小数点ユニット
高性能、ベクトルもサポート
設定項目が多すぎて複雑（私がヨワヨワなだけ）
加算器ツリーを作るなどの用途には向かない

LogiCORE IP Floating-Point Operator

AXIインターフェースが必ず付く
丸めはroundTiesToEvenのみ、subnormalをフラッシュ

単精度浮動小数点数形式

自作してみよう！

まずは浮動小数点数の確認

exponent	significand = 0	significand ≠ 0	value
h00	±zero	subnormal number	$(-1)^{sign} \cdot 2^{-126} \cdot 0.significand$
h01-hFE	normal value		$(-1)^{sign} \cdot 2^{expoent-127} \cdot 1.significand$
hFF	±infinity	NaN	＼

FP32乗算器

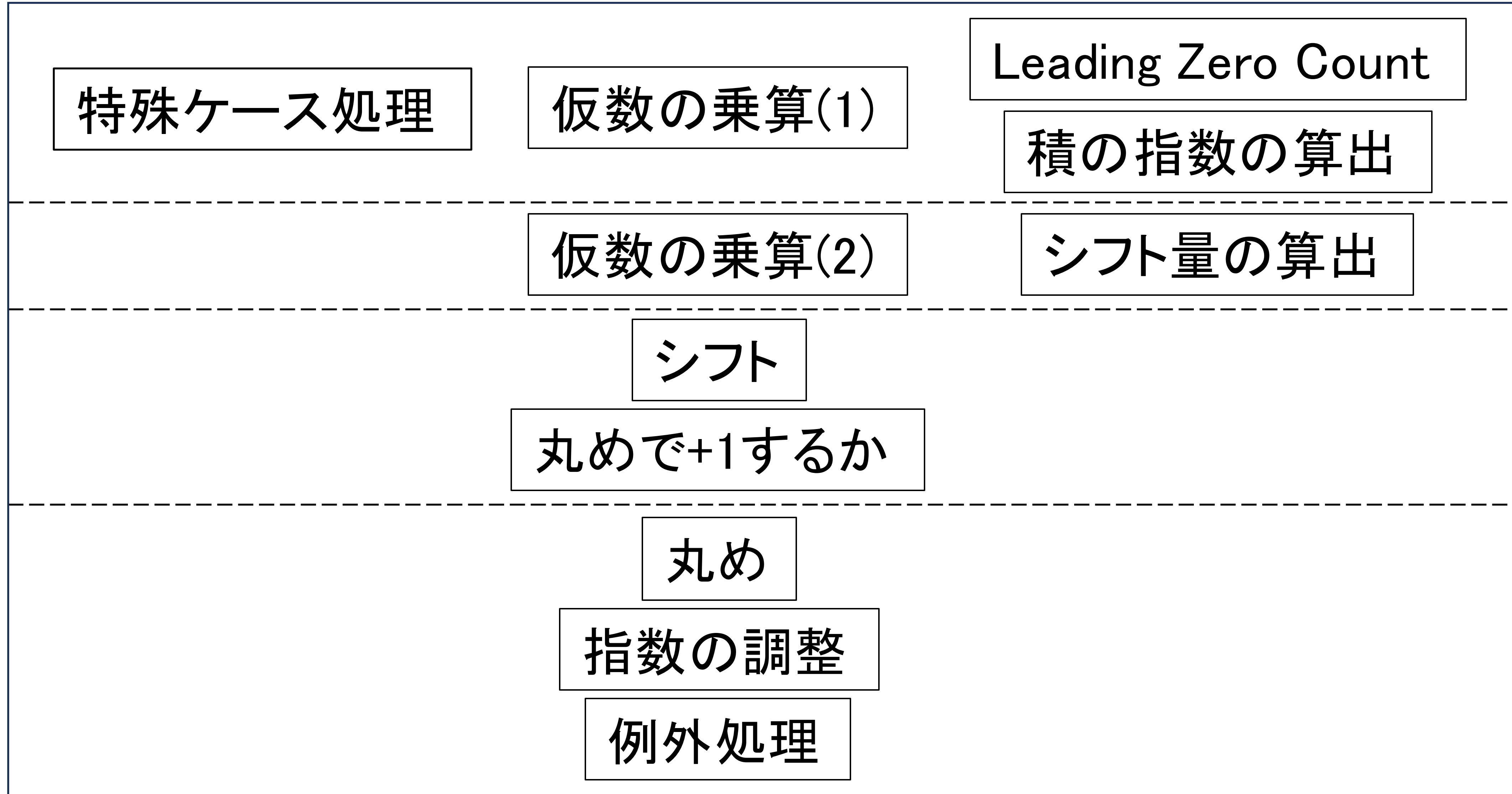
基本方針
○指数部の和と仮数部の積を求める

・指数部には127のゲタがあるため指数部の和から127を引いて積の指数を算出

・指数部が0の非正規仮数は仮数部の積の先頭に0が連続し、積を正規化するためには先頭の0のカウント（Leading Zero Count)や長いシフトが必要

・入力の仮数部の Leading Zero Count を行うことで仮数の乗算と並列にシフト量を算出、指数も調整

FP32乗算器



FP32加算器

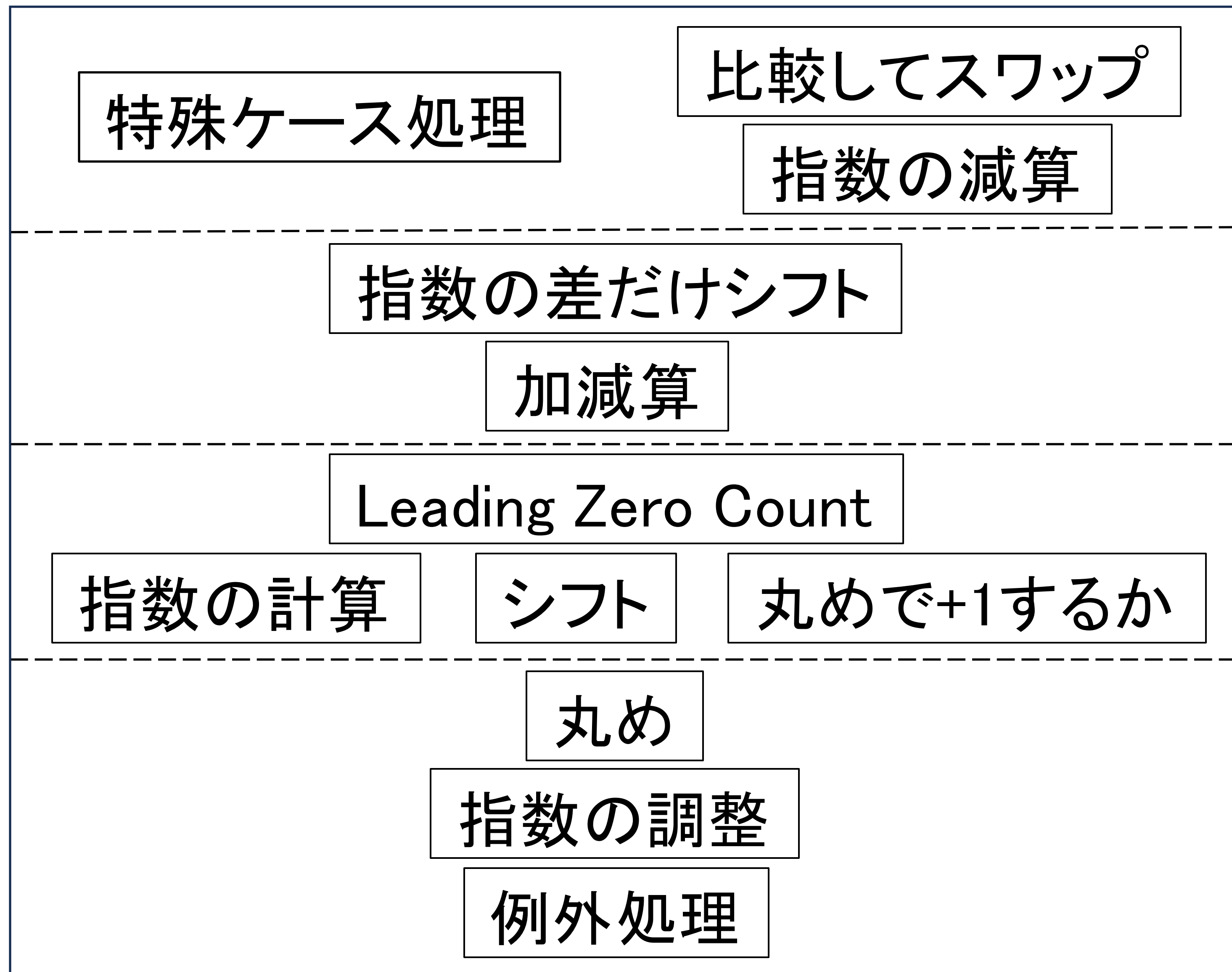
基本方針
○入力の大きさを比較してスワップし、小さい方の入力を指数の差だけ右シフト

・入力の符号部の排他的論理和をとることで実質的な加算か減算か判定

・指数の差が0か1しかない場合は減算で複数bitの桁落ちが発生する可能性があり
Leading Zero Count とシフトが必要

・加算においては 1 bitの桁上がり、減算においては桁下がりが発生する可能性があるため、加減算後の処理を等しくするために加算の入力は1bit右シフト

FP32加算器



合成結果

ZYBO Z7(Zynq-7020)に実装したところ、FP32乗算器とFP32加算器いずれも153MHzで動作
FloPoCoと比較すると、リソースは増えるが速い

	FP32 multiplier (4 stages)			FP32 adder (4 stages)	
	Delay[ns]	LUT	DSP	Delay[ns]	LUT
Mine	6.5	374	2	6.5	554
FloPoCo	7.7	331	1	8.3	414

FloPoCoのversionは4.1.2、 Vivado2024.1にてglobal_retimingをonにして合成

コードは https://github.com/dokunira/floating-point-unit

おまけ

自作RV32I_Zicsrコアを高速化していった話

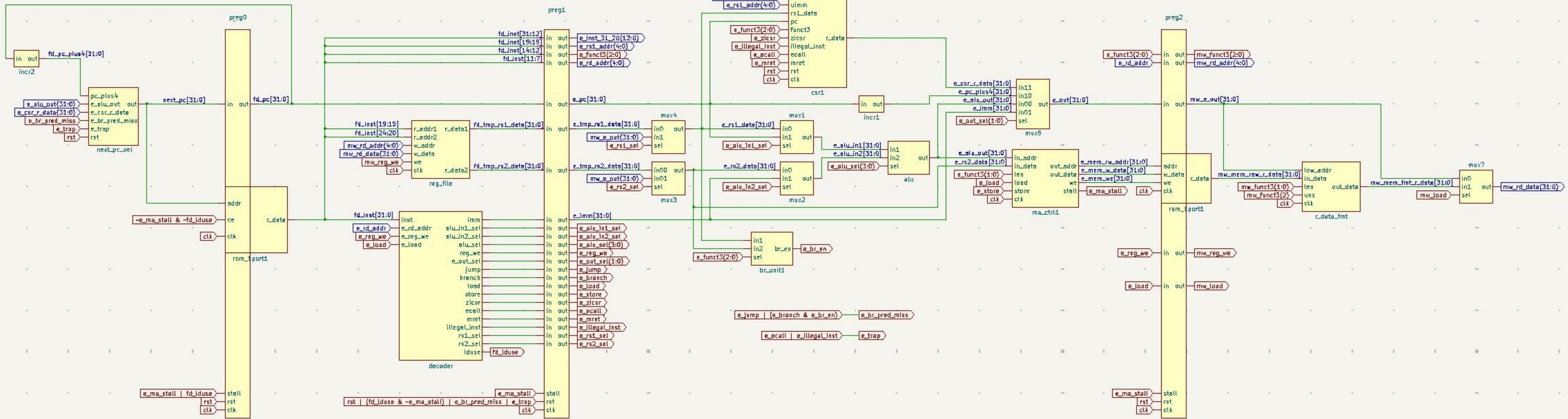
コードは https://github.com/dokunira/rv32i_zicsr_3stage_pipeline

RV32I_Zicsr_3stage_pipeline_v1

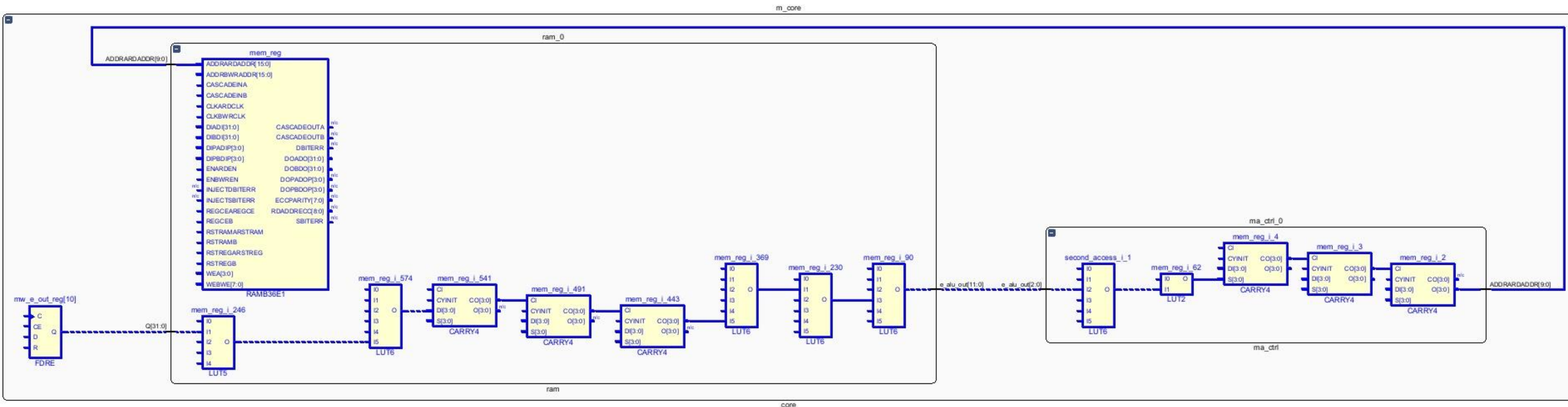
Fetch & Decode

Excute

Memory & Writeback



Implementation (RV32I_Zicsr_3stage_pipeline_v1)



Critical Path Delay : **8.596 ns** (Load , Store)
pipe_reg(FD_to_E) → mux(fw) → mux(alu_in) → alu → ma_ctrl → bram

Implemented Design LUT : **1833**, FF : **1429**

Frequency : **104 MHz**

Memory Access Controller

All RISC-V Load and Store instructions calculate addresses by **adding register and immediate value**.

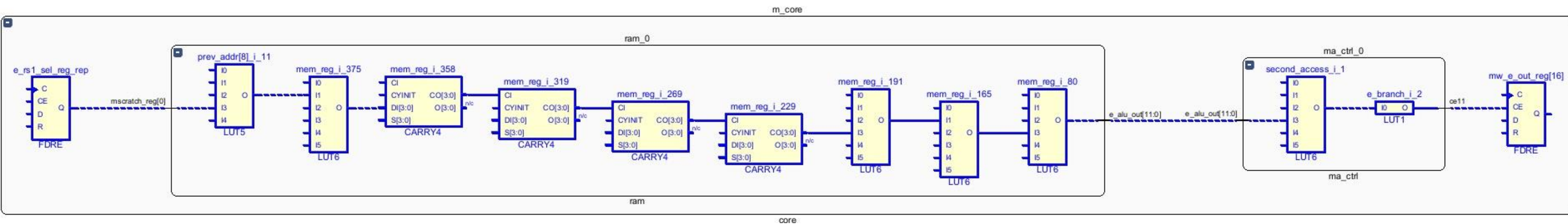
In a misaligned access,
the ALU calculation result is incremented to access a higher address, and then the pipeline is stalled to access a lower address.

I thought this increment was extending the critical path,
thus, I improved it by

- **store the ALU output in a register**
- **use a multiplexer** to select
  - the incremented value of the previous alu output
  - alu output

This ensures that the alu and the incrementer have different paths.

Implementation (RV32I_Zicsr_3stage_pipeline_v2)



Critical Path Delay : **8.625 ns** (Load , Store)

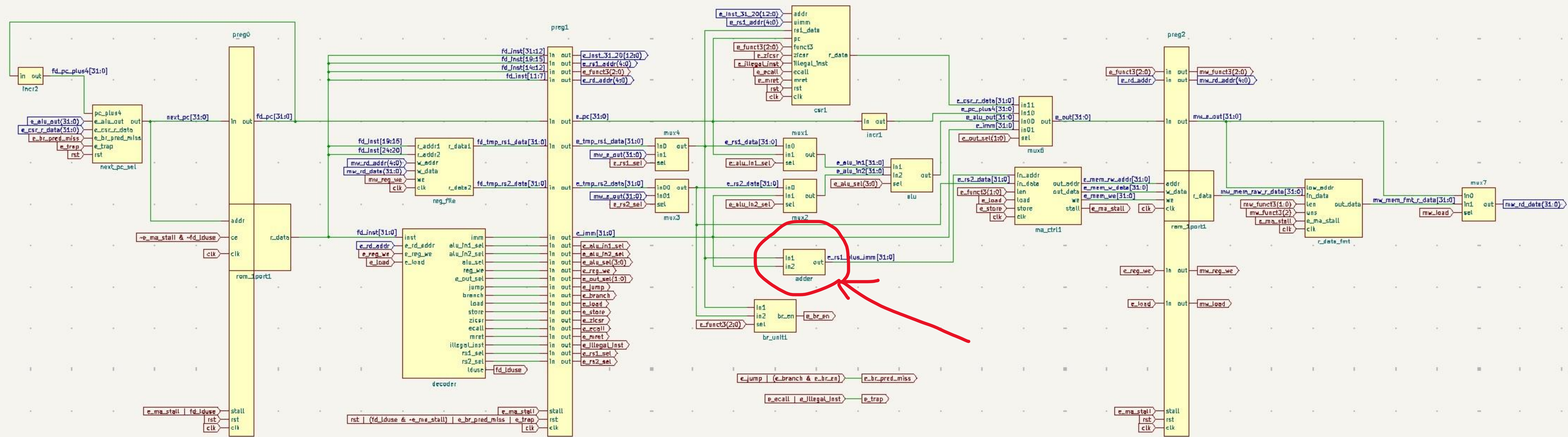
pipe_reg→ mux(fw) → mux(alu_in) → alu → ma_ctrl → pipe_reg(stall)

Implemented Design LUT : **1933**, FF : **1450**

Frequency : **110 MHz**

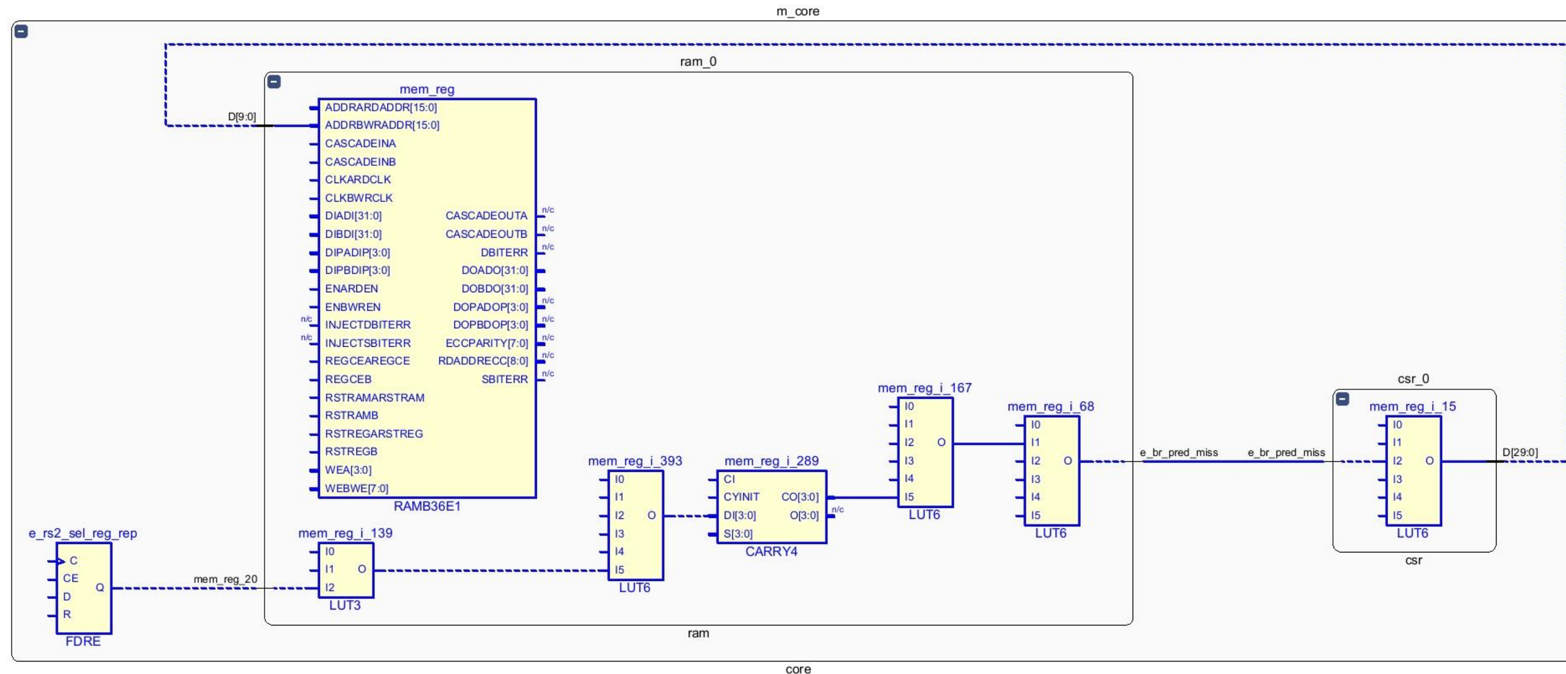
Although the operating speed has increased,
memory access is still on the critical path.

RV32I_Zicsr_3stage_pipeline_v3



To omit the ALU input/output multiplexer from the memory access path,
I added an adder for memory access.

Implementation (RV32I_Zicsr_3stage_pipeline_v3)

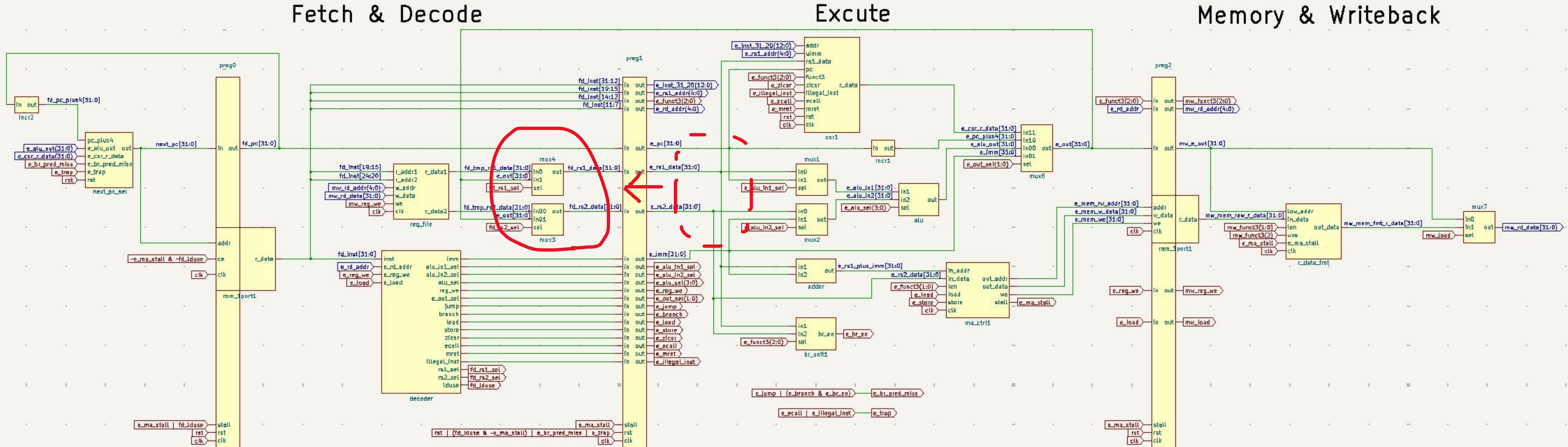


Critical Path Delay : **7.556 ns** (Branch)
pipe_reg→ mux(fw) → br_unit → bram

Implemented Design LUT : **1981**, FF : **1448**

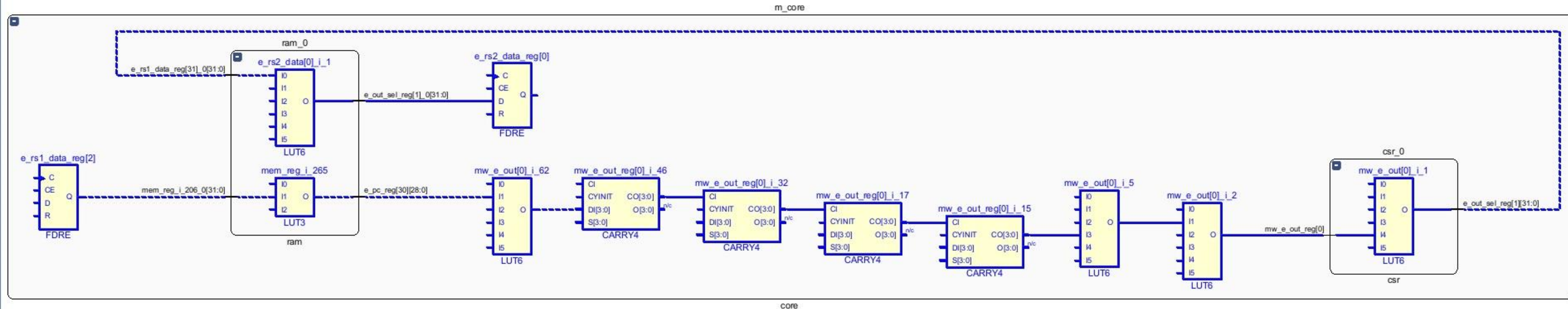
Frequency : **121 MHz**

RV32I_Zicsr_3stage_pipeline_v4



To shorten the branch path, the forwarding was changed from the beginning of the Execute stage to the end of the Decode stage.

Implementation (RV32I_Zicsr_3stage_pipeline_v4)



Critical Path Delay : **7.667 ns** (Forwarding)
pipe_reg→ mux(alu_in) → alu →mux(fw) → pipe_reg

Implemented Design LUT : **1990**, FF : **1445**

Frequency : **126 MHz**