

Comet with Branch Predictor

Team 11

Github: https://github.com/s950449/Comet_With_Branch_Predictor/

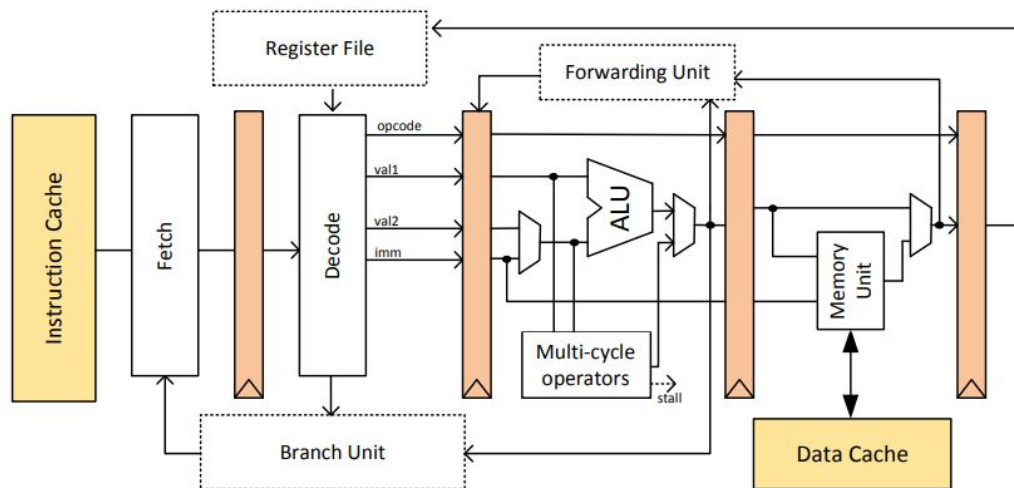
Comet

- Synthesizable RISC-V processor with HLS
- HLS comes with the benefit of simulating and testing processor designs
- RISC-V ISA combined with simple implementation making implementing and tweaking different architectural designs easy

Comet

```
struct FtoDC ftodc;  
struct DCtoEx dctoex;  
struct Extomem extomem;  
struct MemtoWB memtowb;  
while true do  
  ftodc_temp = fetch();  
  dctoex_temp = decode(ftodc);  
  extomem_temp = execute(dctoex);  
  memtowb_temp = memory(extomem);  
  writeback(memtowb);  
  /* -- Handling stalls -- */  
  bool stall[5] = stallLogic();  
  if !stall[0] then  
    | ftodc = ftodc_temp;  
  end  
  if !stall[1] then  
    | dctoex = dctoex_temp;  
  end  
  ...  
  /* -- Handling forwarding -- */  
  bool forward = forwardLogic();  
  if forward then  
    | dctoex.value1 = extomem.result;  
  end  
end
```

Algorithm 2: High-level specification of an explicitly pipelined simulator.



Textbook 5-stage pipelined machine with forwarding and stalling for data hazards, control hazards and multi-cycle arithmetic ops

Comet, but in Catapult

- The camera-ready version of Comet is written in Catapult, which has different HLS rules and different definitions for arbitrary precision data types
- The design is straightforward and can be easily expanded upon
- However, it's somewhat nontrivial to refactor to vivado synthesizable code

Combined Design

- Luckily, there is an [older implementation in Vivado HLS](#)
- The older version is synthesizable but the overall structure is not great
- So, we manage to combine the two to have the best of both worlds

Combined Design

```
=====
== Performance Estimates
=====
+ Timing:
  * Summary:
  +-----+-----+-----+-----+
  | Clock | Target | Estimated | Uncertainty |
  +-----+-----+-----+-----+
  | ap_clk | 8.50 ns | 6.999 ns | 1.06 ns |
  +-----+-----+-----+-----+

+ Latency:
  * Summary:
  +-----+-----+-----+-----+-----+-----+
  | Latency (cycles) | Latency (absolute) | Interval | Pipeline |
  | min | max | min | max | min | max | Type |
  +-----+-----+-----+-----+-----+-----+
  | 442370 | 573440 | 3.760 ms | 4.874 ms | 442370 | 573440 | none |
  +-----+-----+-----+-----+-----+-----+

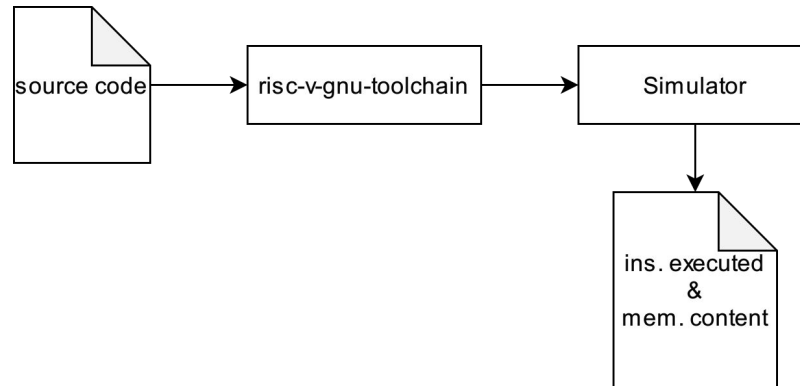
+ Detail:
  * Instance:
  +-----+-----+-----+-----+-----+-----+-----+
  | Instance | Module | Latency (cycles) | Latency (absolute) | Interval | Pipeline |
  | min | max | min | max | min | max | Type |
  +-----+-----+-----+-----+-----+-----+-----+
  | grp_doCycle_fu_487 | doCycle | 4 | 6 | 34.000 ns | 51.000 ns | 4 | 6 | none |
  +-----+-----+-----+-----+-----+-----+-----+

  * Loop:
  +-----+-----+-----+-----+-----+-----+-----+
  | Latency (cycles) | Iteration | Initiation Interval | Trip |
  | min | max | Latency | achieved | target | Count | Pipelined |
  +-----+-----+-----+-----+-----+-----+-----+
  | Loop 1 | 4 | 4 | 1 | - | - | 4 | no |
  | Loop 2 | 24576 | 24576 | 3 | - | - | 8192 | no |
  | Loop 3 | 393210 | 524280 | 6 ~ 8 | - | - | 65535 | no |
  | Loop 4 | 24576 | 24576 | 3 | - | - | 8192 | no |
  +-----+-----+-----+-----+-----+-----+-----+
```

Synthesis Report for the Design

Verification

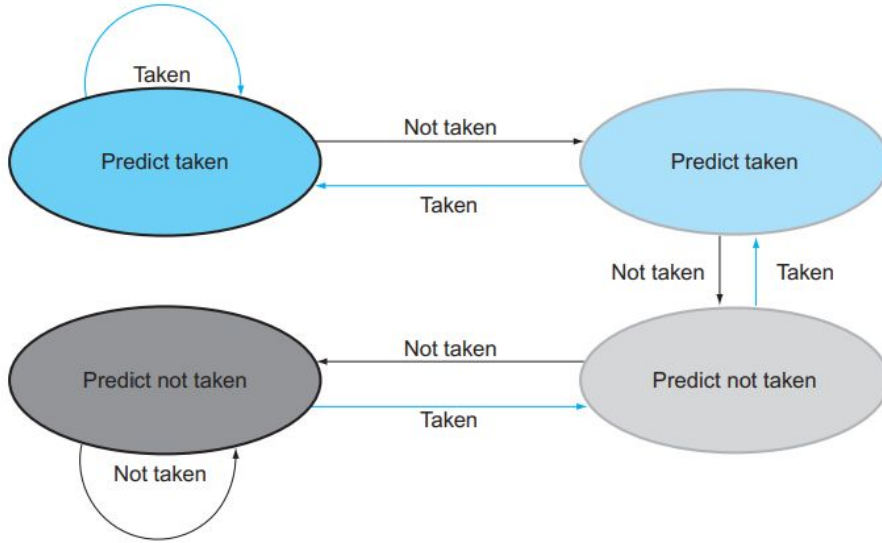
- It's hard to check every instruction executed and values of each register every cycle
- Instead, verification is done by simulating the design with artificial programs, matrix multiplication and Quick Sort, and analyzing memory content after execution



Branch Prediction

- Control hazards in pipelined machine cause pipeline flushes, but there are many branches that exhibit regular patterns
- Branch prediction enables many aggressive execution optimizations

Branch Prediction: N-Bit Predictor



2-Bit Predictor

```
template <int BITS, int ENTRIES>
class BitBranchPredictor : public BranchPredictorWrapper<BitBranchPredictor<BITS, ENTRIES>> {
    static const int LOG_ENTRIES = log2const<ENTRIES>::value;
    static const int NT_START = (1 << BITS) - 1;
    static const int NT_FINAL = (1 << BITS) >> 1;
    static const int T_START = 0;
    static const int T_FINAL = NT_FINAL - 1;

    CORE_UINT(BITS) table[ENTRIES];

public:
    BitBranchPredictor()
    {
        for (int i = 0; i < ENTRIES; i++) {
            table[i] = T_START;
        }
    }

    void _update(CORE_UINT(32) pc, bool isBranch)
    {
        CORE_UINT(LOG_ENTRIES) index = pc.SLC(LOG_ENTRIES, 2);

        if (isBranch) {
            table[index] -= table[index] != T_START ? 1 : 0;
        } else {
            table[index] += table[index] != NT_START ? 1 : 0;
        }
    }

    void _process(CORE_UINT(32) pc, bool& isBranch)
    {
        CORE_UINT(LOG_ENTRIES) index = pc.SLC(LOG_ENTRIES, 2);
        isBranch = table[index] <= T_FINAL;
    }
};
```

N-Bit Predictor in HLS

Branch Prediction: Correlating Branch Predictor

- Correlating Branch Predictor takes outcomes of previous branches into account based on the observation that branches may be correlated
- It's also really easy to implement with N-Bit Branch Predictor

```
template<int CORRELATION_BITS, int PREDICTOR_BITS, int ENTRIES>
class CorrelatingPredictor : public BranchPredictorWrapper<CorrelatingPredictor<CORRELATION_BITS, PREDICTOR_BITS, ENTRIES> > {
    static const int NUM_PREDICTORS = (1 << CORRELATION_BITS);

    BitBranchPredictor<PREDICTOR_BITS, ENTRIES> bp[NUM_PREDICTORS];
    CORE_UINT(CORRELATION_BITS) bhr;

public:
    CorrelatingPredictor() {
        for(int i = 0; i < CORRELATION_BITS; i++) {
            bhr[i] = 0;
        }
    }

    void _update(CORE_UINT(32) pc, bool isBranch) {
        bp[(int)bhr]._update(pc, isBranch);
        bhr = ((bhr << 1) | isBranch);
    }

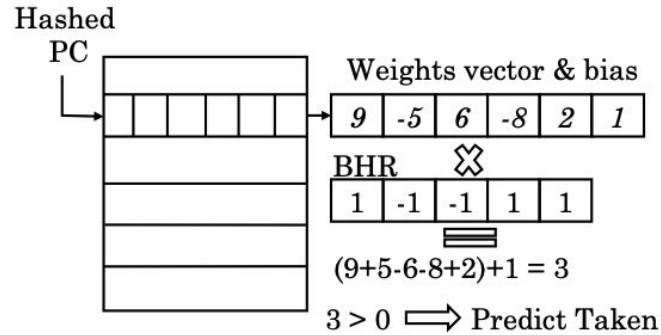
    void _process(CORE_UINT(32) pc, bool& isBranch) {
        bp[(int)bhr]._process(pc, isBranch);
    }
};
```

Branch Prediction: Random Predictor

- Just a pure guess(use rand())
- Easy to implement
- Aware of overhead of this type of branch predictor

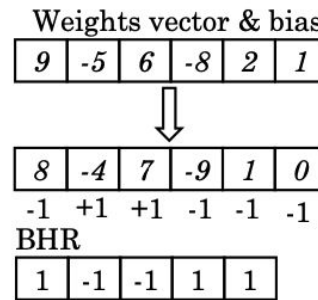
Test/Type	no branch predictor	random branch predictor
dct	17033481	16810980
dijkstra	420505	423406
matmul	56294	60215
qsort	41103	43775

Branch Prediction: Perceptron Predictor

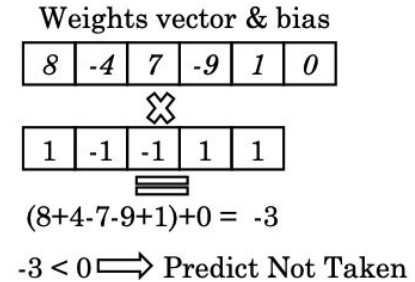


(a) First time prediction

If outcome = not taken



(b) Retraining

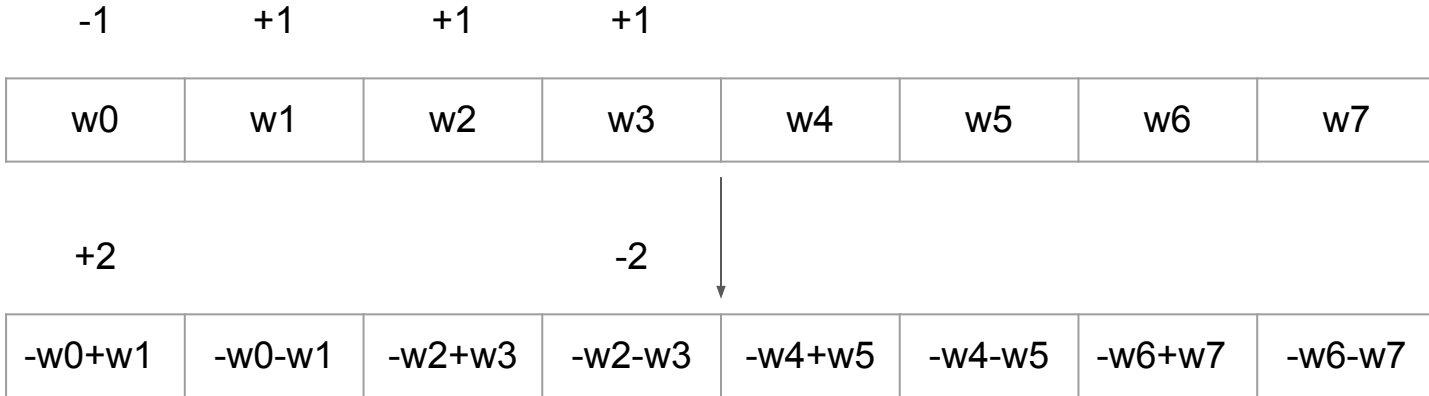


(c) Next time prediction

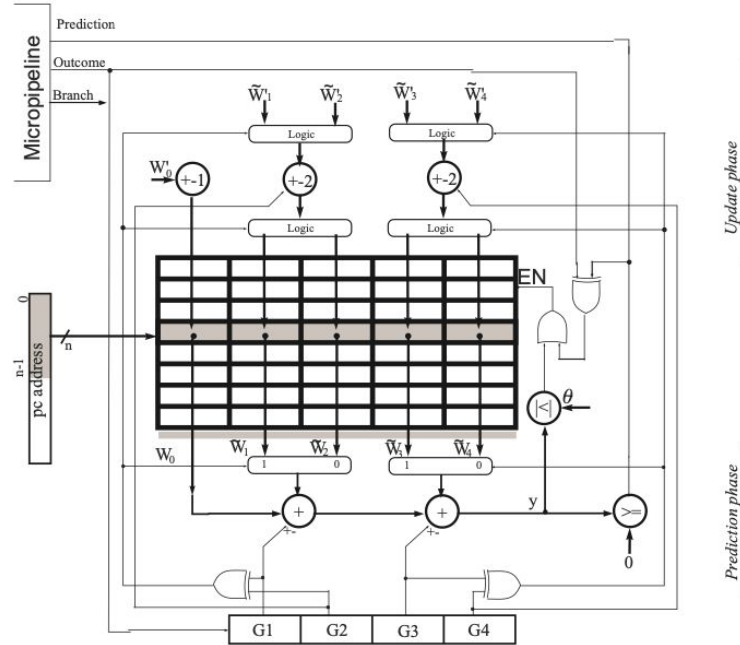
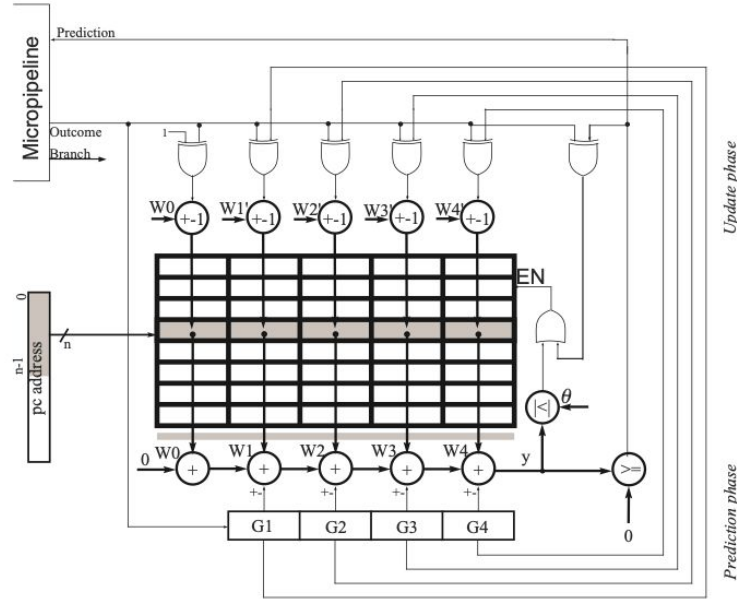
Branch Prediction: Perceptron Predictor



Branch Prediction: Perceptron Predictor



Branch Prediction: Perceptron Predictor



Branch Prediction: Perceptron Predictor

```
dp = perceptron[index][SIZE];

ap_int<BITS> weight[SIZE/2];
ap_int<2> sign[SIZE/2];
#pragma HLS array_partition variable=perceptron dim=2 complete
#pragma HLS array_partition variable=bht          dim=1 complete
#pragma HLS array_partition variable=weight      dim=1 complete
#pragma HLS array_partition variable=sign        dim=1 complete

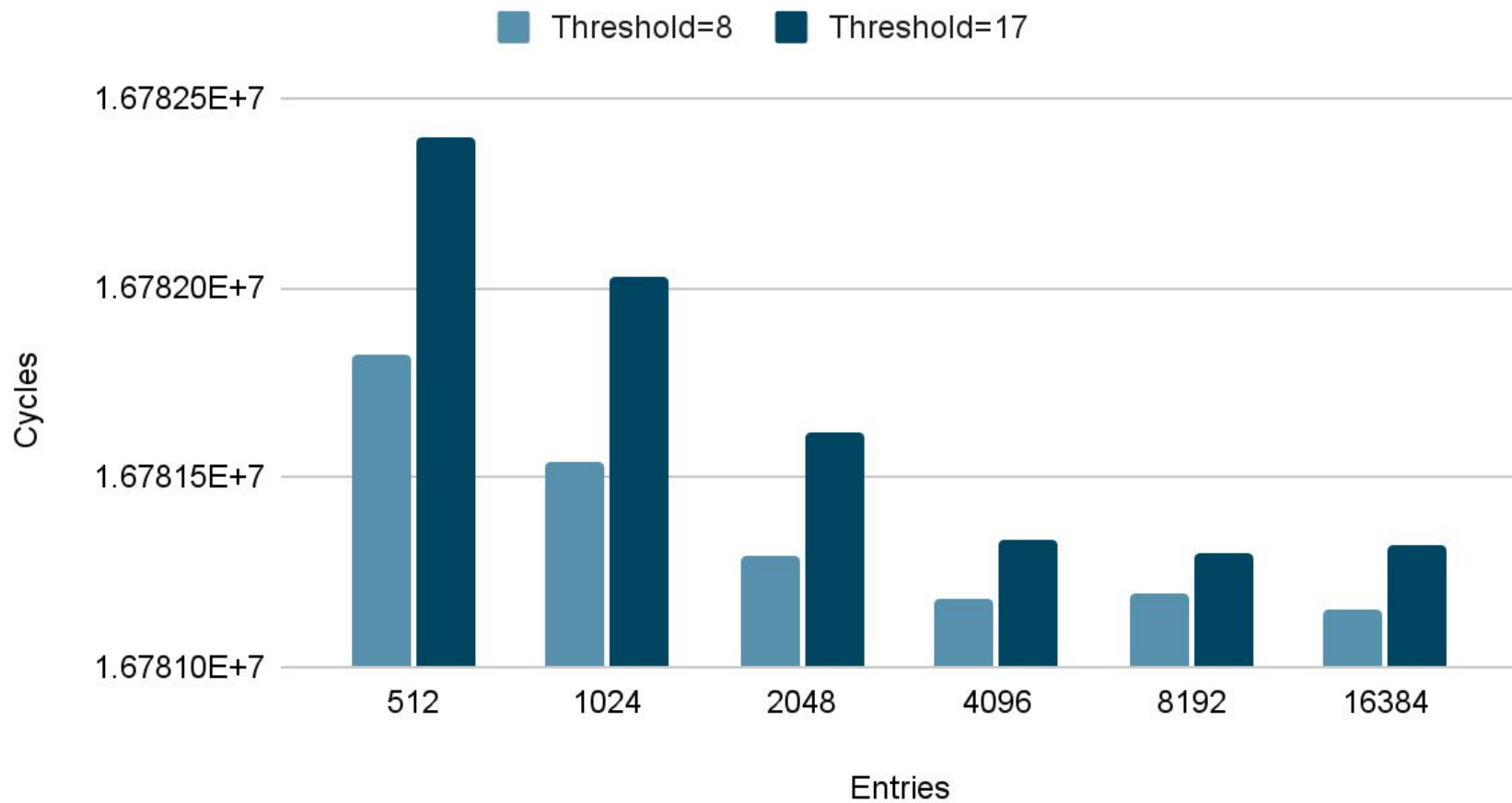
for (int i = 0; i < SIZE/2; i++) {
#pragma HLS PIPELINE
    weight[i] = (bht[i*2] == bht[i*2+1]) ? perceptron[index][i*2+1] : perceptron[index][i*2];
}

for (int i = 0; i < SIZE/2; i++) {
#pragma HLS PIPELINE
    sign[i] = bht[i*2] ? -1 : 1;
}

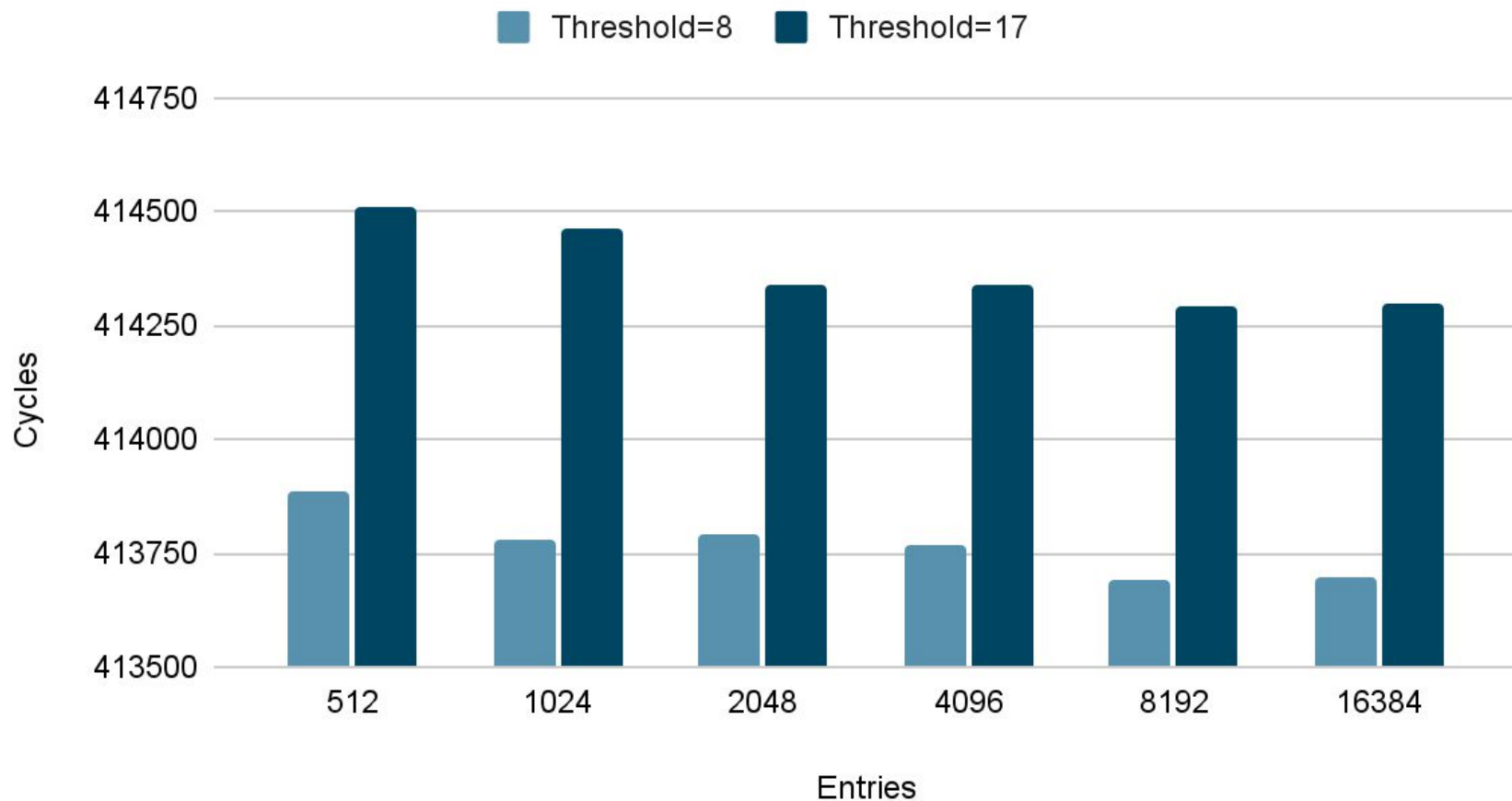
for (int i = 0; i < SIZE/2; i++) {
#pragma HLS UNROLL
    dp += weight[i] * sign[i];
}

pd = dp >= 0;
```

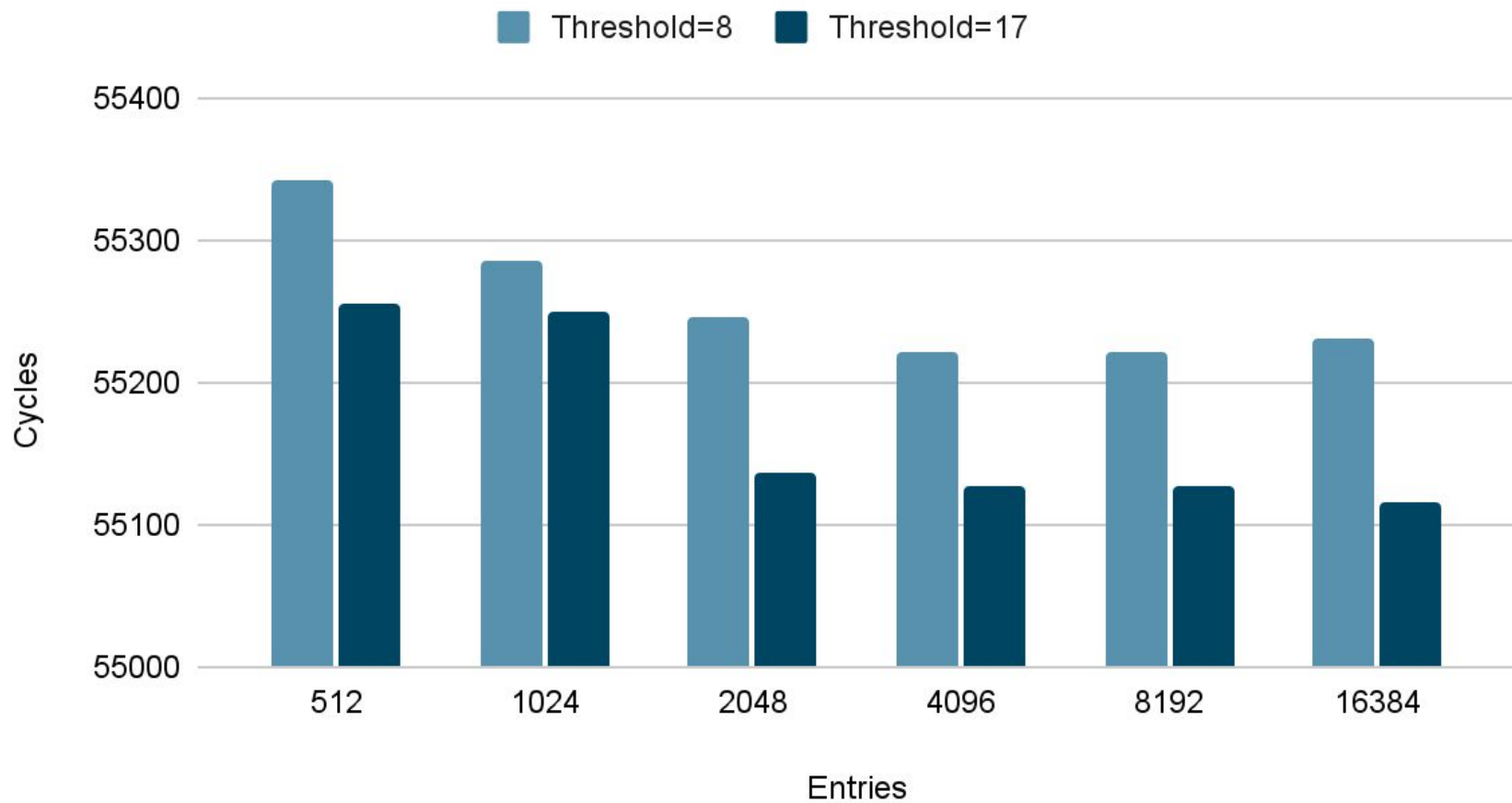

dct,perceptron



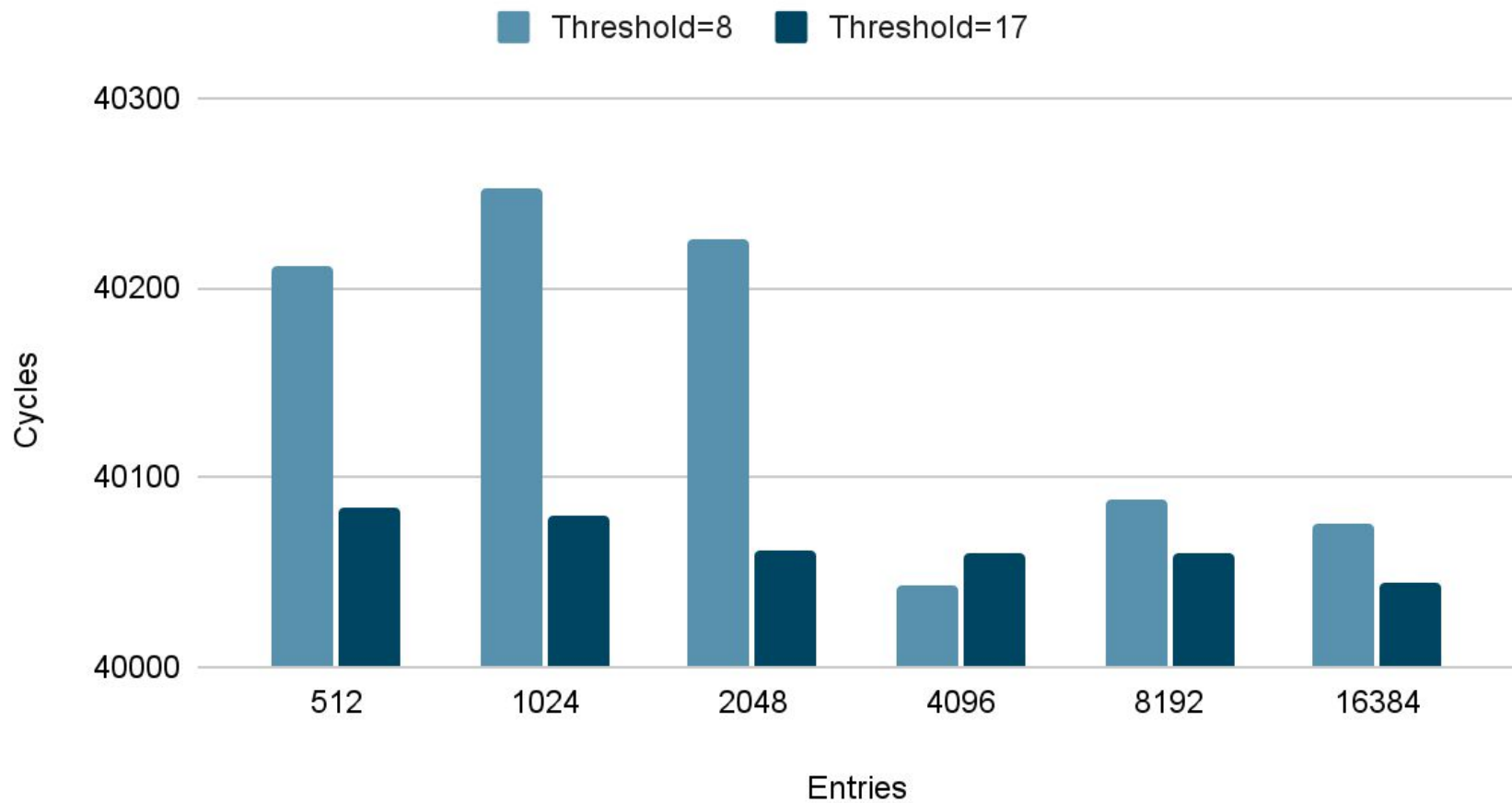
dijkstra,perceptron



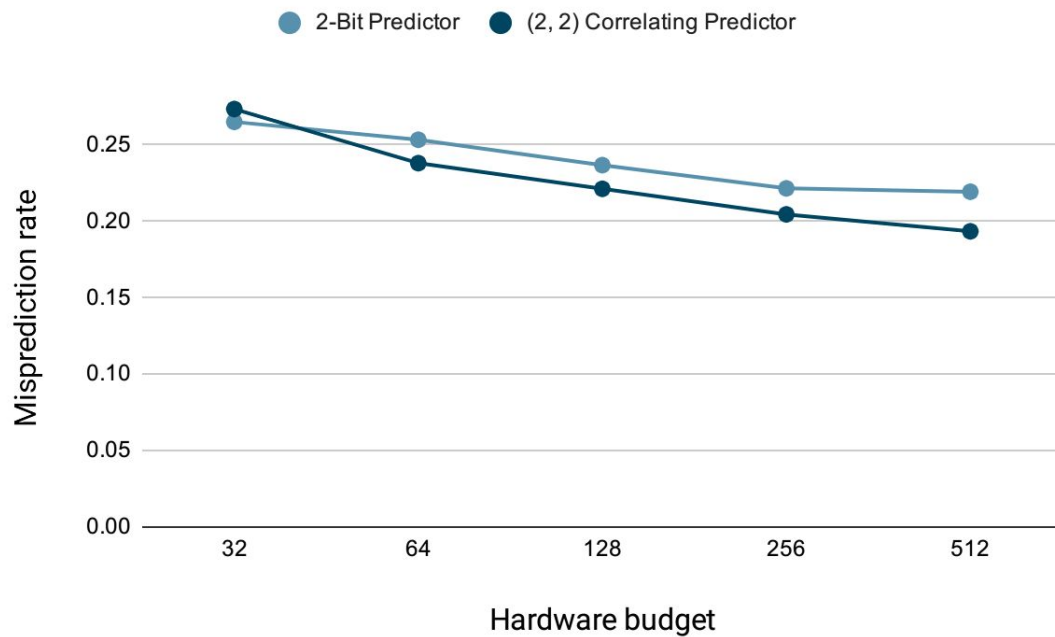
matmul,perceptron



qsort, perceptron

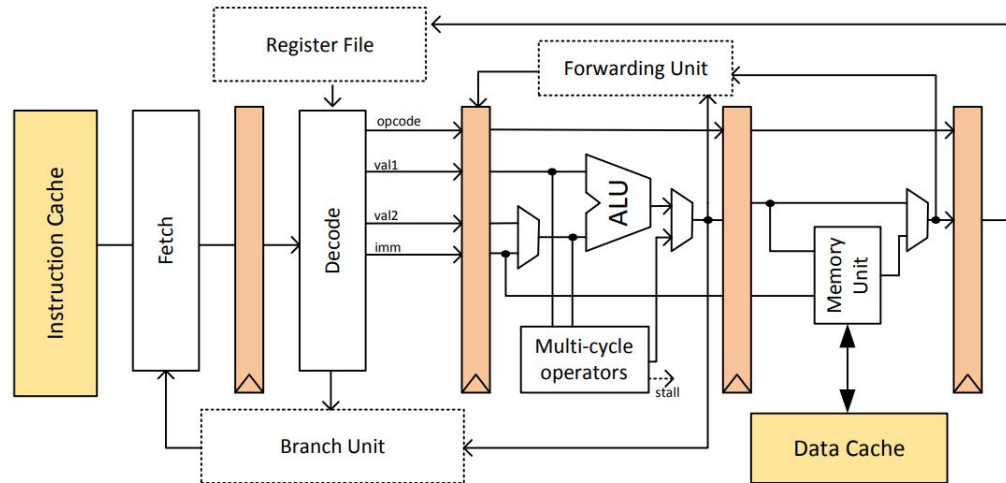


Evaluation



Branch Predictor on Top of Comet

- Branch outcome was originally computed in the EX stage, but we move it to the second stage, ID
- We implement branch prediction by changing the proper pipeline registers, and updating the predictor when the outcome is computed



Export IP and Other Problems

- We can synthesize and export RISC-V Core as a Xilinx IP
- However, we can't easily build a real RISC-V processor on FGPA
- We need to combine memory controller, memory, I/O,etc.
- Although we found a [repo](#) which can generate bitstream on PYNQ
- We should replace original picorv32 core with Comet
 - Different input and output port
 - Whole design more complicated
 - Vivado version not meeted
- So we only use simulator

Q&A

Reference

1. [What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications](#)
2. [A new organization for a perceptron-based branch predictor and its FPGA implementation](#)
3. [RISC-V CPU in HLS](#)
4. [HL5: A 32-bit RISC-V Processor Designed with High-Level Synthesis](#)
5. [Two-Level Adaptive Training Branch Prediction](#)
6. [RISC-V-On-PYNQ](#)