ESE5320: System-on-a-Chip Architecture

Day 18: Nov. 1, 2023

Maps, Associative Memories, Hash Tables

Today

- Associative Memories on FPGAs (Part 1)
- Software Map Trees (Part 2)
- Hash Tables (Part 3)
  - Software
  - Hardware (FPGA) Hash Maps

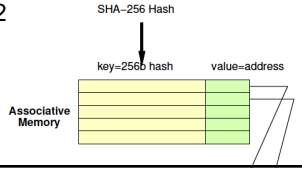
Message

- Many options for Maps
- Hash tables are useful tools

Day 16 Review

Associative Memory

- Maps from a key to a value
  - *associates* a value with a key
- Key not necessarily dense
  - Contrast simple RAM
  - Cannot afford 2



Associative Memories

- Use for deduplication
- Also may use in LZW to reduce BRAMs
  - Just saw
    - **Problem:** Simple 2D tree table requires too many BRAMs
    - **Opportunity:** Tree table sparse

Associative Memories FPGA

Part 1

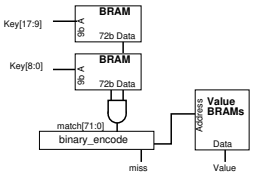
FPGA

- Has BRAMs – normal memories, not associative
- 36Kb BRAM
  - 512x72
- Can be 9b key → 72b value assoc.
  - Just using the memory sparsely
- Or interpret as programmable decoder with 72 match lines

Assoc. Mem from BRAM

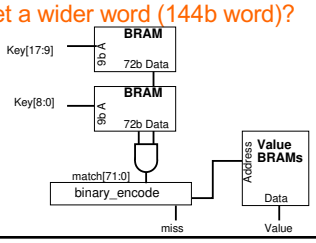
For wider match

- Cover 9b of key with each BRAM
- Use 72 output bits to indicate if one of 72 entries match
- AND together corresponding entries
- Get 72 match bits
- Re-encode match bits to lookup value

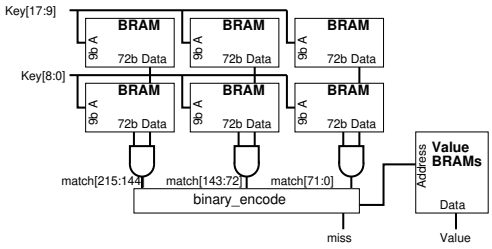


BRAM Associative Memory

- Previous slide expands match width
- How would we expand capacity?
  - Hint: how get a wider word (144b word)?

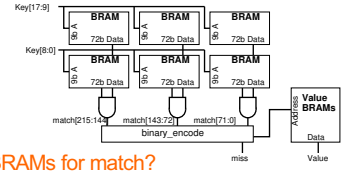


BRAM Associative Memory



Associative Memory Cost

- Match unit
  - Requires 1 BRAM per 9b of key per 72 entries
  - [keylen/9b] ✖ [entries/72]
  - Asymptotically optimal (keylen*entries)
    - But large constants
  - LZW
    - 4K entries
    - 20b key
    - How many BRAMs for match?

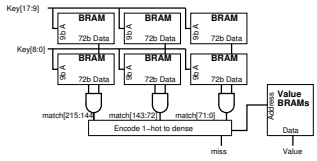


4K LZW Chunk Search: Fully associative

- Match BRAMs:
  - Match key: 20b
  - Entries: 4096
- Value BRAMS:
  - 12b (state [position])
  - 12b x 4096 entries
  - Takes 2 BRAMs

Example Stored Values

Key[17:9]	Key[8:0]	Value
0x001	0x014	0x01
0x001	0x01	0x34
0x0F0	0x014	0xE3
0x0C8	0x113	0xCC



13

Memory Contents

Key[17:9] Match BRAM

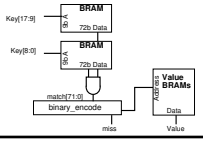
Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	1
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	0	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

Key[8:0] Match BRAM

Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	0
0x014	0	0	0	0	0	1	0	1
0x0C8	0	0	0	0	0	0	0	0
0x0F0	0	0	0	0	0	0	0	0
0x113	0	0	0	0	1	0	0	0

Value BRAM

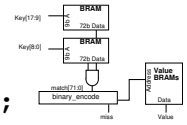
Addr	Value
0x00	0x01
0x01	0x34
0x02	0xE3
0x03	0xCC
0x04	
0x05	
0x06	



(only show bottom 8 b; rest 0's)

14

Code Snippet



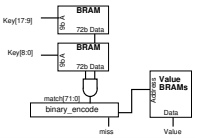

```
ap_uint<72> key_low[512];
ap_uint<72> key_high[512];
int value[72];

match_low=key_low[key%512];
match_high=key_high[(key>>9)%512];
match=match_low & match_high;
addr=binary_encode(match);
res=value[addr];
```


15

How Lookup Work?

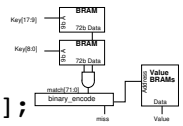
Key[17:9]	Key[8:0]	Value
0x001	0x014	0x01
0x001	0x01	0x34
0x0F0	0x014	0xE3
0x0C8	0x113	0xCC

Lookup 0x214 = 0x001 0x014



16

Code Snippet




```
ap_uint<72> key_low[512];
ap_uint<712> key_high[512];
int value[72];

match_low=key_low[key%512];
match_high=key_high[(key>>9)%512];
match=match_low & match_high;
addr=binary_encode(match);
res=value[addr];
```


17

Memory Contents

Key[17:9] Match BRAM

Addr	7	6	5	4	3	2	1	0
0x001	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	0	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

```
match_low=key_low[key%512];
match_high=key_high[(key>>9)%512]
```

Key[8:0] Match BRAM

What match_low, match_high?

Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	0
0x014	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**
0x0C8	0	0	0	0	0	0	0	0
0x0F0	0	0	0	0	0	0	0	0
0x113	0	0	0	0	1	0	0	0

(only show bottom 8 b; rest 0's)

18

Memory Contents

Key[17:9] Match BRAM

Addr	7	6	5	4	3	2	1	0
0x001	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	0	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

```
match_low=key_low[key%512];
match_high=key_high[(key>>9)%512]

match=match_low & match_high;
```

What match?

Key[8:0] Match BRAM

Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	0
0x014	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**
0x0C8	0	0	0	0	0	0	0	0
0x0F0	0	0	0	0	0	0	0	0
0x113	0	0	0	0	1	0	0	0

(only show bottom 8 b; rest 0's)

19

What does binary_encode do?

binary_encode(0000000…010000)=0x40

- 10000…0 → 71
- 0000…01 → 0
- 0000…010 → 1
- for (i=0<i<72;i++)
  - If (bit[i]==1) return i
- Return(MISS); // if not find (i.e., all 0's)
- Technicalities – maybe check only one 1

20

Memory Contents

Key[17:9] Match BRAM

Addr	7	6	5	4	3	2	1	0
0x001	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	0	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

```
match_low=key_low[key%512];
match_high=key_high[(key>>9)%512]

match=match_low & match_high;

addr=binary_encode(match);
```

What addr?

Key[8:0] Match BRAM

Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	0
0x014	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**
0x0C8	0	0	0	0	0	0	0	0
0x0F0	0	0	0	0	0	0	0	0
0x113	0	0	0	0	1	0	0	0

(only show bottom 8 b; rest 0's)

21

Memory Contents

Key[17:9] Match BRAM

Addr	7	6	5	4	3	2	1	0
0x001	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	0	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

Key[8:0] Match BRAM

Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	0
0x014	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**
0x0C8	0	0	0	0	0	0	0	0
0x0F0	0	0	0	0	0	0	0	0
0x113	0	0	0	0	1	0	0	0

Value BRAM

Addr	Value
0x00	0x01
0x01	0x34
0x02	0xE3
0x03	0xCC
0x04	
0x05	
0x06	

```
res=value[addr];
```

What res?

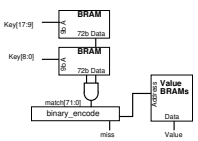
(only show bottom 8 b; rest 0's)

22

How Lookup Work?

Key[17:9]	Key[8:0]	Value
0x001	0x014	0x01
0x001	0x01	0x34
0x0F0	0x014	0xE3
0x0C8	0x113	0xCC

Lookup 0x1E14 = 0x0F0 0x014



23

Memory Contents

Key[17:9] Match BRAM

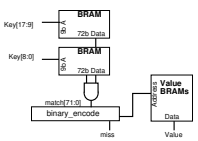
Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	1
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	0	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

Key[8:0] Match BRAM

Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	0
0x014	0	0	0	0	0	1	0	1
0x0C8	0	0	0	0	0	0	0	0
0x0F0	0	0	0	0	0	0	0	0
0x113	0	0	0	0	1	0	0	0

Value BRAM

Addr	Value
0x00	0x01
0x01	0x34
0x02	0xE3
0x03	0xCC
0x04	
0x05	
0x06	



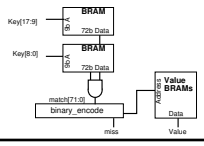
(only show bottom 8 b; rest 0's)

24

Add another entry

match	Key[17:9]	Key[8:0]	Value
0	0x001	0x014	0x01
1	0x001	0x01	0x34
2	0x0F0	0x014	0xE3
3	0x0C8	0x113	0xCC
4	0x0C8	0x01	0x2B

How BRAM contents change to add this entry for 0x19001



25

Memory Contents

Key[17:9] Match BRAM

Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	1
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	0	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

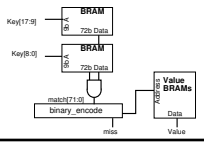
Value BRAM

Addr	Value
0x00	0x01
0x01	0x34
0x02	0xE3
0x03	0xCC
0x04	
0x05	
0x06	

Key[8:0] Match BRAM

Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	0
0x014	0	0	0	0	0	1	0	1
0x0C8	0	0	0	0	0	0	0	0
0x0F0	0	0	0	0	0	0	0	0
0x113	0	0	0	0	1	0	0	0

(only show bottom 8 b; rest 0's)



26

Memory Contents

Key[17:9] Match BRAM

Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	1
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	1	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

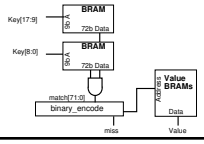
Value BRAM

Addr	Value
0x00	0x01
0x01	0x34
0x02	0xE3
0x03	0xCC
0x04	0x2B
0x05	
0x06	

Key[8:0] Match BRAM

Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	1	0	0	1	0
0x014	0	0	0	0	0	1	0	1
0x0C8	0	0	0	0	0	0	0	0
0x0F0	0	0	0	0	0	0	0	0
0x113	0	0	0	0	1	0	0	0

(only show bottom 8 b; rest 0's)



27

4K Chunk LZW Search

	BRAMs	Operations/byte
Brute Search	1	4K
Tree with Dense RAM	384	1
Tree with Full Assoc	173	1

36Kb BRAMs on ZU3EG = 216

28

Checkpoint

- Notice levels of mapping:
  - Prefix Tree algorithm
  - Formulated on a 2D memory
  - Then implemented in assoc. memory
    - (later with Tree … hash table)

29

Software Map

Part 2

30

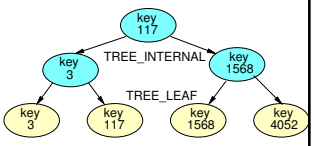
Software Map

- Map abstraction
  - void insert(key,value);
  - value lookup(key);
- Will typically have many different implementations

Preclass 2

- For a capacity of 4096
- How many memory accesses needed
  - When lookup fail?
  - When lookup succeed (on average)?

Tree Map (Preclass 4)



- Build search tree
- Walk down tree
- For a capacity of 4096, assume balanced…
- How many tree nodes visited
  - When lookup fail?
  - When lookup succeed (on average)?

Tree Map LZW

- Each character requires $\log_2$(dict) lookups
  - 12 for 4096
- Each internal tree node hold
  - Key (20b for LZW), value (12b), and 2 pointers (12b)
  - 7B
- Total nodes 4K*2
- Need 14 BRAMs for 4K chunk

Tree Insert

- Need to maintain balance
- Doable with O(log(N)) insert
  - Tricky
  - See Red-Black Tree
    - https://en.wikipedia.org/wiki/Red–black_tree
    - https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

4K Chunk LZW Search Story so far….

	BRAMs	Operations/byte
Brute Search	1	4K
Tree with Dense RAM	384	1
Tree with Full Assoc	173	1
Tree with Tree	14	12

36Kb BRAMs on ZU3EG = 216

Hash Tables

Part 3

37

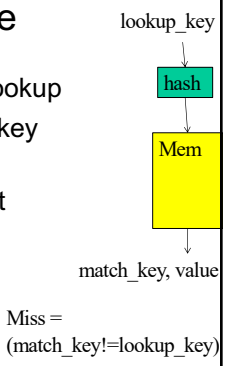
High Performance Map

- Would prefer not to search
- Want to do better than $\log_2(N)$ time
  - $\log_2(N)$ achieved with tree
- Direct lookup in arrays (memory) is good…

38

Hash Table

- Attempt to turn into direct lookup
- Compute some function of key
  - A hash
- Perform lookup at that point
- If hash maps a single entry (or no entry)
  - Great, got direct lookup
    - Like sparse table case



39

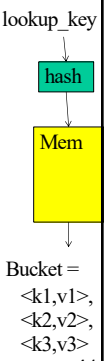
Hash Entries

- Average number of entries per hash when N > HASH_CAPACITY?
  - Concrete example
    - N= 4096
    - HASH_CAPACITY=256

40

Hash Table

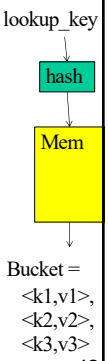
- Attempt to turn into direct lookup
- Compute some function of key
  - A hash
- Perform lookup at that point
- Typically, prepared for several keys to map to same hash → call it a bucket
  - Keep list or tree of things in each bucket



41

Hash Table

- Compute some function of key
  - A hash
- Perform lookup at that point
- Find bucket with small number of entries
  - Searching that bucket easier
  - …but no absolute guarantee on maximum bucket size

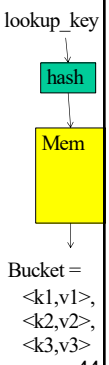


42

Hardware Hash Tables

Hardware Hash

- Want to avoid variable size buckets
  - So can read in one lookup
    - Can make wider for some fixed number of slots
  - So can resolve in one cycle



Hash Size Distribution

- Look at what the distribution looks like for number of entries
- N – number of entries
- C – HASH_CAPACITY (N<C) [buckets]
- m – number of items mapped to a bucket
- Compute distribution for each bucket size (m)

Preclass 3

N=1024

m→	0	1	2	3	4+
C=1024					
C=2048		0.30			
C=4096					

$$\binom{N}{m}\left(\frac{1}{C}\right)^{m}\left(1-\frac{1}{C}\right)^{N-m}$$

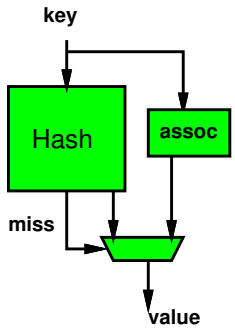
Hash

- Can tune hash parameters to control distribution
- Spend more memory → smaller buckets → less work finding things in buckets
  - Memory-Time tradeoff
- Still have possibility of large buckets
  - …but probability is low

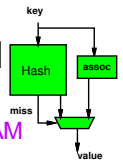
Idea

- Hash mostly works
- Engineer hash to hold most cases
  - Combination of
    - sparcity (entries>N)
    - Hold multiple entries per hash value
- Few cases that overflow
  - Store in small fully associative memory

Hybrid Hash+Assoc.



LZW 4K Chunk Hybrid



- 72 entry assoc. match
  - needs 3 match BRAMs + 1 value BRAM
  - Associative match 20b key
  - 72 entries (72/4096=1.7% for 4096)
- So, can hold ~1% conflicts in 4K hash
- Hash N=4096, C=16384, m=2, store 2
  - Prob 3+: <1% (see table 1024, 4096)
  - 20b key+12b value=4B per entry
  - 16384*2*4B=4*2*4 BRAMs
- 32+4=36 BRAMs

Further Optimization

- Previous example illustrative
  - Not necessarily optimal (explore parameters)
    - Expect not optimal
- May be able to do better with multiple hashes
  - See Dhawan reading paper
  - May need to use that design in hybrid configuration with assoc. memory like previous example

Allow Imperfect?

- **Question:** impact on compression if cannot store a few tree entries?
- Some encodings will find shorter matches than optimal
- **Q:** Impact on compression rate as a function of conflict rate?
- How compare to compression rate impact of chunk size?
  - Larger chunk with conflict rate vs. smaller chunk with smaller (or no) conflict rate
- → another tradeoff to explore

Hash Complexity

- Want to compute these lookup hashes for hardware fast
  - In a single cycle to keep II down for LZW
  - Can xor-together a set of bits quickly in hardware
    - Any 6-bits for one output bit in a single 6-LUT
    - Means capacity must be power-of-2

4K Chunk LZW Search

	BRAMs	Operations/byte
Brute Search	1	4K
Tree with Dense RAM	384	1
Tree with Full Assoc	173	1
Tree with Tree	14	12
Tree with Hybrid Hash	36	1

36Kb BRAMs on ZU3EG = 216

Big Ideas

- Many ways to implement maps
- Near O(1) Map access → Hash Table

Admin

- Feedback
- Reading for Monday on web
- First project milestone due Friday
  - Including teaming
- P2 out