ESE5320: System-on-a-Chip Architecture

Day 17: Oct. 31, 2022

LZW, Associative Maps

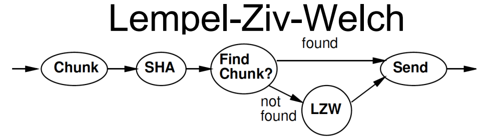
Today

- LZW Compression (Part 1)
- Associative Memory (Part 2)
  - Custom
  - FPGA
- Software Maps
  - Tree (Part 3)

Message

- Can adaptively compress data using recurring substrings
- Rich design space for Maps

Part 1: LZW Compression Lempel-Ziv-Welch



Day 16 Reminder

Idea

- Use data already sent as the dictionary
  - Give short names to things in dictionary
  - Don't need to pre-arrange dictionary
  - Adapt to common phrases/idioms in a particular document

Day 16 Reminder

Encoding

- Greedy simplification
  - Encode by successively selecting the longest match between the head of the remaining string to send and the current window

Day 16 Reminder

Algorithm Concept

- While data to send
  - Find largest match in window of data sent
  - If length too small (length=1)
    - Send character
  - Else
    - Send <x,y> = <match-pos,length>
    - (actually, send dictionary entry)
  - Add data encoded into sent window
  - Expand dictionary

Day 16: Preclass 7

- How many comparisons per invocation?

```
#define DICT_SIZE 4096
#define LENGTH 256
// clen<=LENGTH
int longest_match(char dict[DICT_SIZE], char candidate[LENGTH], int clen) {
  int best_len=0;
  int best_loc=-1;
  for (int i=0;i<DICT_SIZE-clen;i++) {
    j=0;
    while((candidate[j]==dict[i+j]) && (j<clen))  {
      j++;
      }
    if (j>best_len) {
       best_len=j;
       best_loc=i;
       }
    }
  return((best_loc<<8)|best_len);
}
```


Idea

- Avoid O(Dictionary-size) work
  - Only need to match against positions that start with the character(s) in string to encode
    - Separate dictionary for each?

0	1	2	3	4	5	6	7	8	9	10
I		A	M		S					

Only check position 0 for “starts with I”

Idea

- Avoid O(Dictionary-size) work
  - Only need to match against positions that start with the character(s) in string to encode
    - Separate dictionary for each?
- T-dictionary:
  - Tap, The, Their, Then, There, Tuesday

Idea

- Avoid O(Dictionary-size) work
  - Only need to match against positions that start with the character(s) in string to encode
    - Separate dictionary for each?
- T-dictionary:
  - Tap, The, Their, Then, There, Tuesday
- If prefix same, why check redundantly?
  - Generalize: Store things with common prefix together
    - Share prefix among substrings
  - Represent all strings as prefix tree

Idea

- Avoid O(Dictionary-size) work
  - Only need to match against positions that start with the character(s) in string to encode
    - Separate dictionary for each?
- If prefix same, why check redundantly?
  - Store things with common prefix together
  - Share prefix among substrings
  - Represent all strings as prefix tree
- Follow prefix trees with fixed work per input character

Tree Algorithm

Tree Root for each character

- Follow tree according to input until no more match
- Send <name of last tree node>
  - Dictionary entry
    - Named sequentially by insertion
- Extend tree (dictionary) with new character
- Start over with this character

I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			

I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc

I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc
A	32	spc	257	spc<-A

I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc
A	32	spc	257	spc<-A
M	65	A	258	A<-M

I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc
A	32	spc	257	spc<-A
M	65	A	258	A<-M
	77	M	259	M<-spc

I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc
A	32	spc	257	spc<-A
M	65	A	258	A<-M
	77	M	259	M<-spc
S	32	spc	260	spc<-S

19

I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc
A	32	spc	257	spc<-A
M	65	A	258	A<-M
	77	M	259	M<-spc
S	32	spc	260	spc<-S
A	83	S	261	S<-A

20

I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc
A	32	spc	257	spc<-A
M	65	A	258	A<-M
	77	M	259	M<-spc
S	32	spc	260	spc<-S
A	83	S	261	S<-A
M	<nothing>			

21

I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc
A	32	spc	257	spc<-A
M	65	A	258	A<-M
	77	M	259	M<-spc
S	32	spc	260	spc<-S
A	83	S	261	S<-A
M	<nothing>			
	258	AM	262	258<-spc

22

I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc
A	32	spc	257	spc<-A
M	65	A	258	A<-M
	77	M	259	M<-spc
S	32	spc	260	spc<-S
A	83	S	261	S<-A
M	<nothing>			
	258	AM	262	258<-spc
S	<nothing>			

23

I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc
A	32	spc	257	spc<-A
M	65	A	258	A<-M
	77	M	259	M<-spc
S	32	spc	260	spc<-S
A	83	S	261	S<-A
M	<nothing>			
	258	AM	262	258<-spc
S	<nothing>			
A	260	spc S	263	260<-A

24

I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc
A	32	spc	257	spc<-A
M	65	A	258	A<-M
	77	M	259	M<-spc
S	32	spc	260	spc<-S
A	83	S	261	S<-A
M	<nothing>			
	258	AM	262	258<-spc
S	<nothing>			
A	260	spc S	263	260<-A
M	<nothing>			

I AM SAM SAM I AM

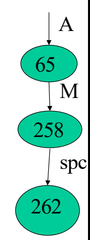
Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc
A	32	spc	257	spc<-A
M	65	A	258	A<-M
	77	M	259	M<-spc
S	32	spc	260	spc<-S
A	83	S	261	S<-A
M	<nothing>			
	258	AM	262	258<-spc
S	<nothing>			
A	260	spc S	263	260<-A
M	<nothing>			
	<nothing>			

I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc
A	32	spc	257	spc<-A
M	65	A	258	A<-M
	77	M	259	M<-spc
S	32	spc	260	spc<-S
A	83	S	261	S<-A
M	<nothing>			
	258	AM	262	258<-spc
S	<nothing>			
A	260	spc S	263	260<-A
M	<nothing>			
	<nothing>			
I	262	AM spc	264	262<-I

Tree Example

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc
A	32	spc	257	spc<-A
M	65	A	258	2A<-M
	77	M	259	M<-spc
S	32	spc	260	spc<-S
A	83	S	261	S<-A
M	<nothing>			
	258	AM	262	258<-spc
S	<nothing>			
A	260	spc S	263	260<-A
M	<nothing>			
	<nothing>			
S	262	AM spc	264	262<-I



Large Memory Implementation

- int encode[SIZE][256];
- Name tree node by insertion order
- c is a character
- Encode[x][c] holds the next tree node that extends tree node x by symbol c
  - Or NONE if there is no such tree node

Memory Tree Algorithm

```
curr=0; // pointer into input chunk
nextdict=NUM_SYMBOLS;
// dict[i]= symbol i for i<NUM_SYMBOLS
while (curr<chunk_size)
  x=input[curr];
  while(encode[x][input[curr]]!=NONE)
     x=encode[x][input[curr]]; curr++;
  send x
  tree[nextdict].parent=x;
  tree[nextdict].sym=input[curr];
  encode[x][input[curr]]=nextdict++;
```

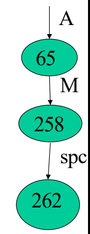
Memory Tree Algorithm

```
curr=0; // pointer into input chunk
nextdict=NUM_SYMBOLS;
// dict[i]= symbol i for i<NUM_SYMBOLS
while (curr<chunk_size)
  x=input[curr];
  while(encode[x][input[curr]]!=NONE)
    x=encode[x][input[curr]]; curr++;
  send x
  tree[nextdict].parent=x;
  tree[nextdict].sym=input[curr];
  encode[x][input[curr]]=nextdict++;
```

Follow Tree

Memory Tree Algorithm

```
curr=0; // pointer into input chunk
nextdict=NUM_SYMBOLS;
// dict[i]= symbol i for i<NUM_SYMBOLS
while (curr<chunk_size)
  x=input[curr];
  while(encode[x][input[curr]]!=NONE)
    x=encode[x][input[curr]]; curr++;
  send x
  tree[nextdict].parent=x;
  tree[nextdict].sym=input[curr];
  encode[x][input[curr]]=nextdict++;
```



I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I<-spc
A	32	spc	257	spc<-A
M	65	A	258	A<-M
	77	M	259	M<-spc
S	32	spc	260	spc<-S
A	83	S	261	S<-A
M	<nothing>			
	258	AM	262	258<-spc
S	<nothing>			
A	260	spc S	263	260<-A
M	<nothing>			
	<nothing>			
I	262	AM spc	264	262<-I

Memory Tree Algorithm

```
curr=0; // pointer into input chunk
nextdict=NUM_SYMBOLS;
// dict[i]= symbol i for i<NUM_SYMBOLS
while (curr<chunk_size)
  x=input[curr];
  while(encode[x][input[curr]]!=NONE)
    x=encode[x][input[curr]]; curr++;
  send x
  tree[nextdict].parent=x;
  tree[nextdict].sym=input[curr];
  encode[x][input[curr]]=nextdict++;
```

How much work per character to encode?
Hint: always go thru tree loop at least once

Map

- In Part 3 will talk about higher-level maps
- Version of encode/decode in
  - Geeks-for-Geeks description
  - Decoder we provide
- Based on Maps of strings
  - Simpler to state
  - Likely not O(1) per symbol
  - More expensive implementation in hardware
    - Version here better for simple hardware

Algorithm with Map

```
curr=0; // pointer into input chunk
nextdict=NUM_SYMBOLS;
// dict initialized symbol i for i<NUM_SYMBOLS
while (curr<chunk_size)
  x=String(input[curr]);
  while(dict.lookup(x+input[curr])!=NONE)
    x=x+input[curr]; curr++;
  send dic.lookup(x);
  dict.insert(x+input[curr],nextdict++);
```

I AM SAM SAM I AM

Input	Send	Dict	Add Entry	Add What
I	<nothing>			
	73	I	256	I spc
A	32	spc	257	spc A
M	65	A	258	AM
	77	M	259	M spc
S	32	spc	260	spc S
A	83	S	261	SA
M	<nothing>			
	258	AM	262	AM spc
S	<nothing>			
A	260	spc S	263	spc SA
M	<nothing>			
	<nothing>			
I	262	AM spc	264	AM spc I

Compact Memory

- int encode[SIZE][256];
- How many entries in this table are not NONE?
  - Hint: SIZE is total number of positions.
    How many characters process?
    Maximum number of insertions per character processed?

Compact Memory

- int encode[SIZE][256];
- Table is very sparse
- If store as associative memory
  - At most SIZE entries
- Look at how to implement associative memories next

LZW So Far – 4KB chunks

- Brute Force
  - Needs one byte per byte = 4KB = 1 BRAM
  - DICT_SIZE=4096 comparisons per byte
- Dense memory encode[SIZE][256]
  - 12b key (1.5B)
  - Need 1.5*4096*256= 384*4KB
    = 384 BRAMs
  - 1 comparison and lookup per byte
  - (maybe should be SIZE+256)

Complexity

- Reminder from Day 15
  - Optimized implementations tend to be more complex
  - Seeing examples of that today…

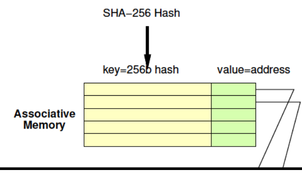
Associative Memories

Part 2

Day 16 Review

Associative Memory

- Maps from a key to a value
- Key not necessarily dense
  - Contrast simple RAM
  - Cannot afford 2^{256} word memory



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

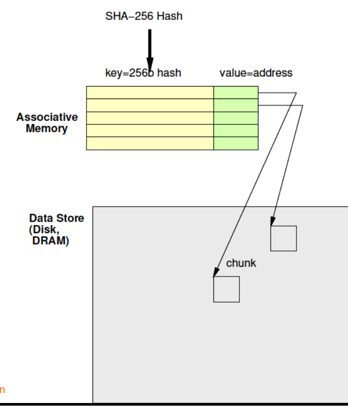
Day 16 Review

Deduplicate

- Compute chunk hash
- Use chunk hash to lookup known chunks
  - Data already have on disk
  - Data already sent to destination, so destination will know
- If lookup yields a chunk with same hash
  - Check if actually equal (maybe)
- If chunks equal
  - Send (or save) pointer to existing chunk

Day 16 Review

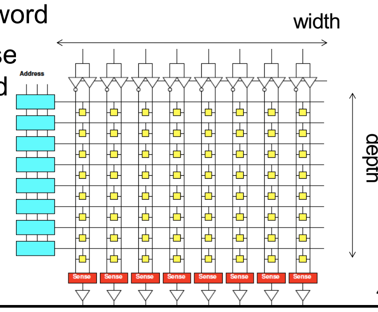
Deduplication Architecture



Custom Hardware Associative Memory

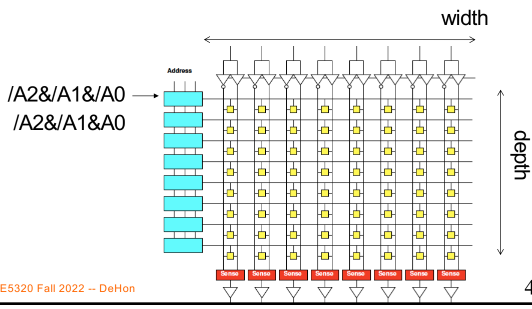
Memory Block Review

- Match on address
- Select wordline for a row
- Reads out a word
- Address dense and hardwired
- One row for each 2^{Abits} values



Address Blocks

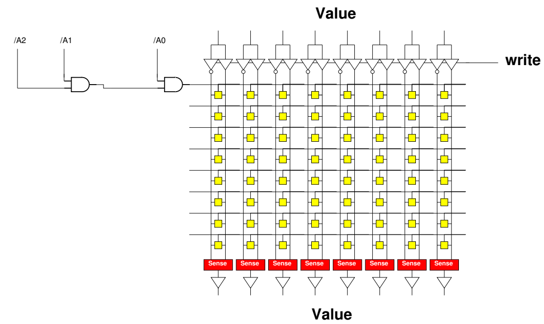
- Each address match is AND



Penn ESE5320 Fall 2022 -- DeHon 49

49

Address Blocks

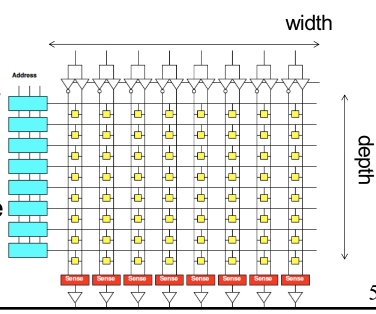


Penn ESE5320 Fall 2022 -- DeHon 50

50

Memory Block Associative

- Want address as key
- Word is value
- Key sparse
- Rows<$2^{keybits}$
- Entries<$2^{keybits}$
- Key programmable

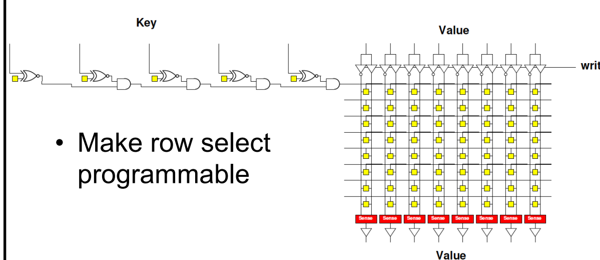


Penn ESE5320 Fall 2022 -- DeHon 51

51

Programmable Key

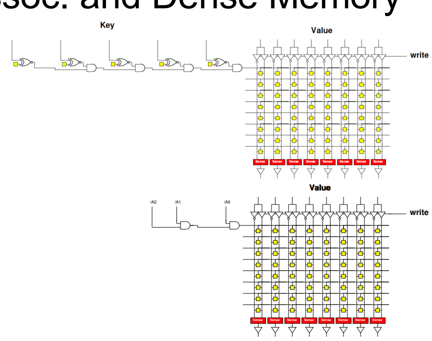
- Make row select programmable



Penn ESE5320 Fall 2022 -- DeHon 52

52

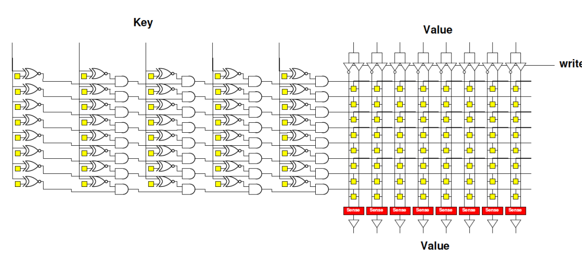
Contrast
Assoc. and Dense Memory



Penn ESE5320 Fall 2022 -- DeHon 53

53

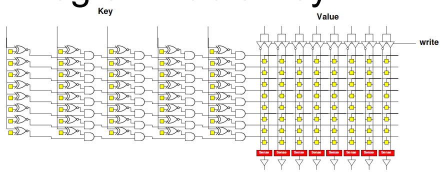
Associative Memory Bank



Penn ESE5320 Fall 2022 -- DeHon 54

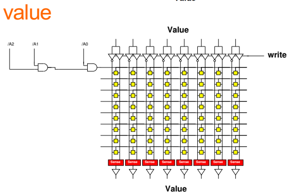
54

Programmable Key



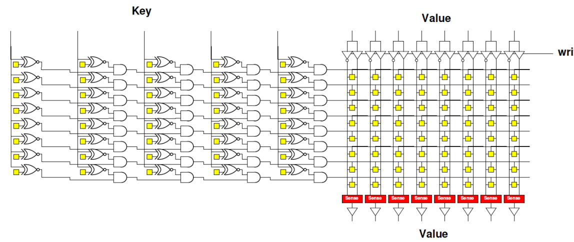
Assoc: 5b key, 8 entries, 8b value
How many memory bits?

Direct: 8 entries, 8b value
How many memory bits?



55

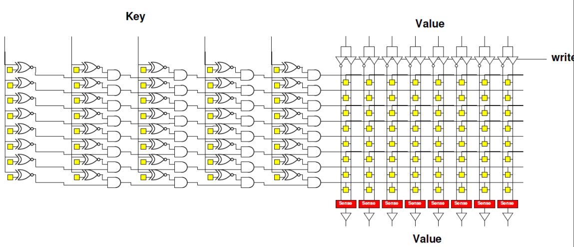
Associative Memory Bank



- Memory cells = entries*(keybits+valuebits)

56

Associative Memory Bank

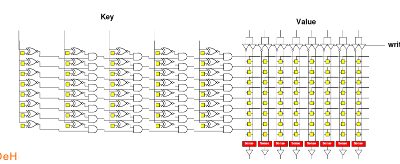


- Will need to be able to write into key
  - Another "fixed" decoder to generate key-word line for programming

57

Associate Memory Cost

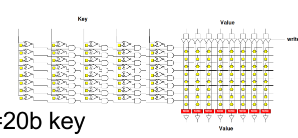
- More expensive than equal capacity SRAM memory bank
  - Memory cells in decoder
  - Need to support write into key



58

Associate Memory Cost

- Physical associative memory for 4KB LZW Chunk tree encode
  - 4K entries
  - 12b (pos) output
  - 12b (pos)+8b (char)=20b key



- Memory cells assoc.?
- Compare direct 4Kx12 memory (cells)?
  - How much larger is assoc. for same capacity?
- Compare 4096*256 with 12b result for dense LZW case (cells)?
  - How much larger to solve same problem

59

FPGA

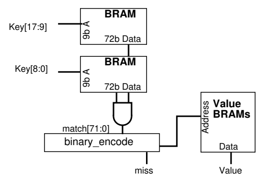
- Has BRAMs – normal memories, not associative
- 36Kb BRAM
  - 512x72
- Can be 9b key → 72b value assoc.
  - Just using the memory sparsely
- Or interpret as programmable decoder with 72 match lines

60

Assoc. Mem from BRAM

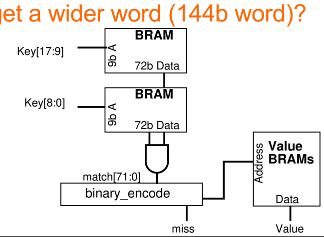
For wider match

- 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

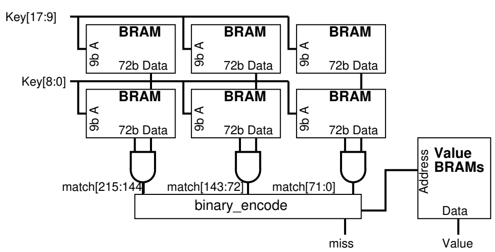


BRAM Associative Memory

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

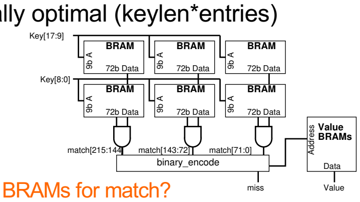


BRAM Associative Memory



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?

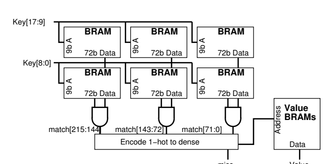


4K LZW Chunk Search: Fully associative

- 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



Memory Contents

Key[17:9] Match BRAM

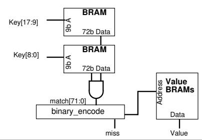
Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	1
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	0	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

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

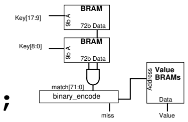
Value BRAM

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



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

Code Snippet



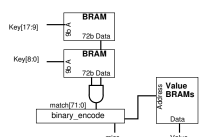

```
ap_uint<72> key_low[512];
ap_uint<72> key_high[512];
int value[72];

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];
```

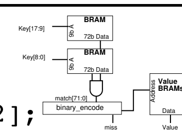

How Lookup Work?

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

Lookup 0x214 = 0x001 0x014



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];
```


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

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

What match_low, match_high?

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

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

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

match=match_low & match_high;
```

What match?

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

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

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);
```

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

What addr?

(only show bottom 8 b; rest 0’s)

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];
```

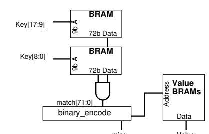
What res?

(only show bottom 8 b; rest 0’s)

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



Memory Contents

Key[17:9] Match BRAM

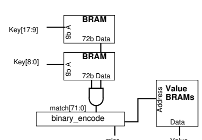
Addr	7	6	5	4	3	2	1	0
0x001	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	0	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

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	

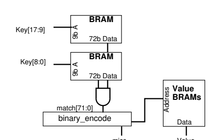


(only show bottom 8 b; rest 0’s)

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



Memory Contents

Key[17:9] Match BRAM

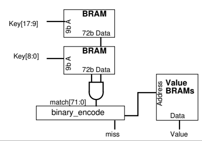
Addr	7	6	5	4	3	2	1	0
0x001	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	0	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

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)

Memory Contents

Key[17:9] Match BRAM

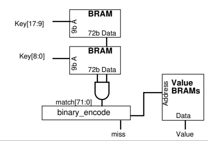
Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	1
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	1	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

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

Value BRAM

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



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

4K Chunk LZW Search

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

36Kb BRAMs on ZU3EG = 216

Checkpoint

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

Lecture Ended Here

Software Map

Part 3

Software Map

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

Penn ESE5320 Fall 2022 -- DeHon 85

85

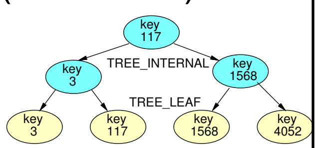
Preclass 1

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

Penn ESE5320 Fall 2022 -- DeHon 86

86

Tree Map (Preclass 2)



- 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)?

Penn ESE5320 Fall 2022 -- DeHon 87

87

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

Penn ESE5320 Fall 2022 -- DeHon 88

88

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/

Penn ESE5320 Fall 2022 -- DeHon 89

89

4K Chunk LZW Search

	BRAMs	Operations
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

Penn ESE5320 Fall 2022 -- DeHon 90

90

Big Ideas

- Can adaptively compress data using recurring substrings
  - With constant work for symbol
- Rich design space for engineering associative map solutions

Admin

- Feedback (including HW7)
- Reading for Wednesday on Web
- First project milestone due Friday
  - Including teaming