ESE532:
System-on-a-Chip Architecture

Day 17: Nov. 2, 2020

Associative Maps, Hash Tables

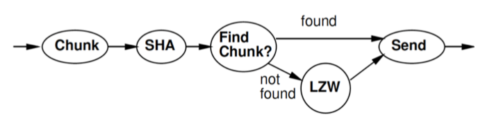
Today

- LZW Compression (Part 1)
- Associative Memory (Part 2)
  - Custom
  - FPGA
- Software Maps
  - Tree (Part 3)
  - Hash Tables (Part 4)
- Hardware (FPGA) Hash Maps (Part 5)
  - (probably next time)

Message

- Rich design space for Maps
- Hash tables are useful tools

Part 5:
LZW Compression



Preclass 1

- I AM S<2,3><5,4><0,4>

- Message?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I		A	M		S											

Preclass 2, 3

- Bits in unencoded (decoded) message?
  - Assume 8b char
- Bits for encoded message?
  - Assume 9b for character
    - 1 bit to say is a character, then 8b char
  - And 9b for <x,y> pair
    - 1 bit char, 4b for each of x and y

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

Encoding

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

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>
  - Add data encoded into sent window

Preclass 4

- 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; 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

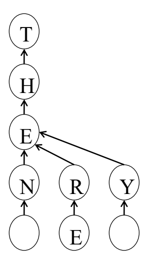
- Avoid O(Dictionary-size) work
  - Only need to match against positions that start with the character(s) in string to encode
    - Separate dictionary for each?
- T-dictionary:
  - Tap, The, Their, Then, There, Tuesday
- If prefix same, why check redundantly?
  - Generalize: Store things with common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

- THEN AND THERE, THEY STOOD…



Tree Algorithm

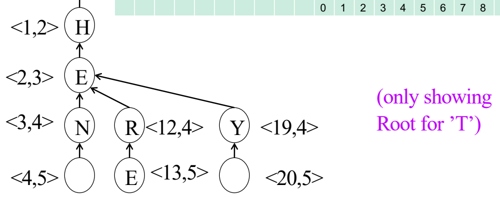
Tree Root for each character

- Follow tree according to input until no more match
- Send <name of last tree node>
  - An <x,y> pair
- Extend tree with new character
- Start over with this character

Tree Example

- Label with <lastpos,len> pair
- THEN AND THERE, THEY STOOD…

T	H	E	N		A	N	D		T	H	E	R	E	,		T	H	E	Y
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19



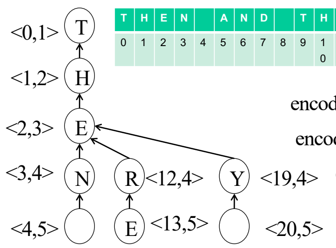
Large Memory Implementation

- int encode[SIZE][256];
- Name tree node by position in chunk
  - lastpos
- c is a character
- Encode[lastpos][c] holds the next tree node that extends tree node lastpos by c
  - Or NONE if there is no such tree node

Tree Example

(only showing Root for 'T')

- Label with <lastpos,len> pair
- THEN AND THERE, THEY STOOD…



T	H	E	N		A	N	D		T	H	E	R	E	,		T	H	E	Y
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

encode[2]['N']=3

encode[2]['R']=12

encode[2]['Y']=19

encode[2]['A']=NONE

Large Table for Tree

Addr	spc	A	…	D	E	…	H	…	N	…	R	S	T	…	Y
-1		5											0		
0							1								
1					2										
2									3		12				19
3	4														
4															
5									6						
6				7											
7	8														
8															
9															
10															
12															

Memory Tree Algorithm

```
curr – pointer into input chunk
// follow tree
y=0; x=-1; // for no match, yet…
while(encode[x][input[curr+y]]!=NONE)
    x=encode[x][input[curr+y]]; y++;
If (y>0)
  send <x,y>
else
  send input[curr+y]
encode[x][input[curr+y]]=curr+y
curr=curr+y+1
```


Memory Tree Algorithm

```
curr – pointer into input chunk
// follow tree
y=0; x=-1; // for no match, yet…
while(encode[x][input[curr+y]]!=NONE)
    x=encode[x][input[curr+y]]; y++;
If (y>0)
  send <x,y>
else
  send input[curr+y]
encode[x][input[curr+y]]=curr+y
curr=curr+y+1
```

Follow Tree

Memory Tree Algorithm

```
curr – pointer into input chunk
// follow tree
y=0; x=-1; // for no match, yet.
while(encode[x][input[curr+y]]!=NONE)
    x=encode[x][input[curr+y]]; y++;
If (y>0)
  send <x,y>
else
  send input[curr+y]
encode[x][input[curr+y]]=curr+y
curr=curr+y+1
```

How much work per character to encode?
Hint:
1) match case
2) not match

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?
    How many 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]
  - Need 2*256B per byte = 512*4KB
    = 512 BRAMs
  - 1 comparison and lookup per byte

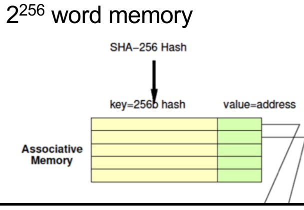
Associative Memories

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



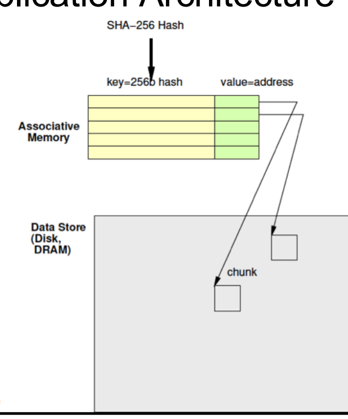
Day 16 Review

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

Deduplication Architecture

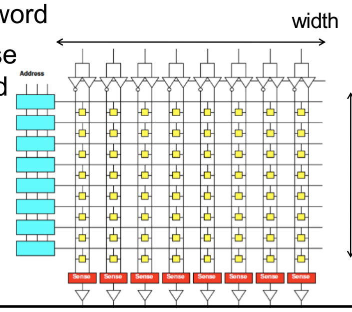
Day 16 Review



Custom Hardware Associative Memory

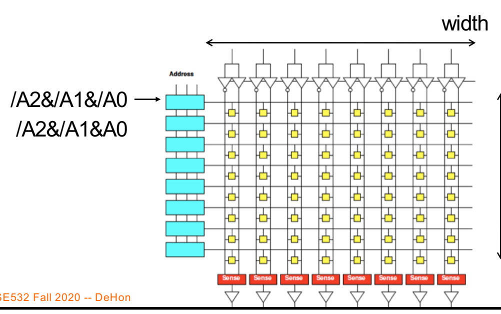
Memory Block Review

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

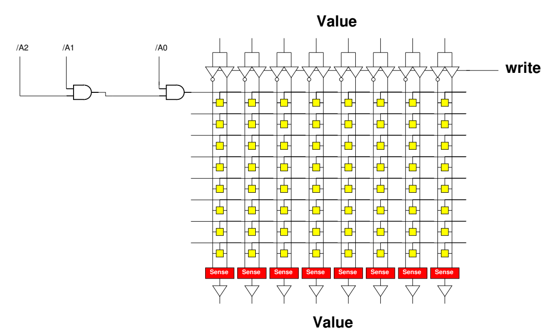


Address Blocks

- Each address match is AND

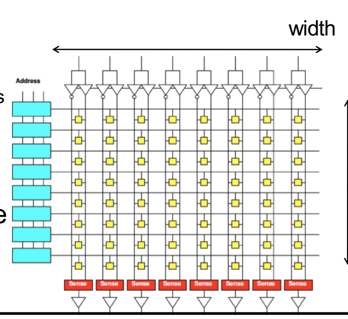


Address Blocks



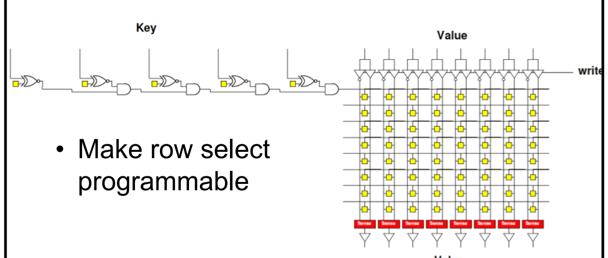
Memory Block Associative

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

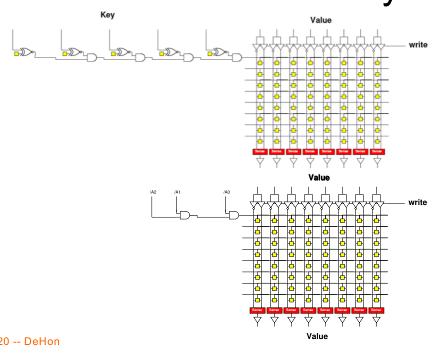


Programmable Key

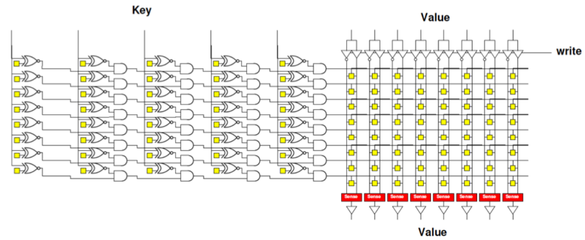
- Make row select programmable



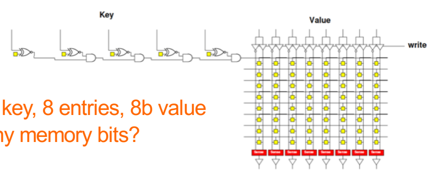
Contrast
Assoc. and Dense Memory



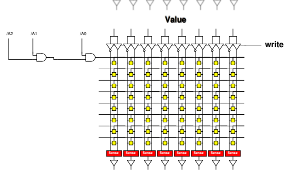
Associative Memory Bank



Programmable Key

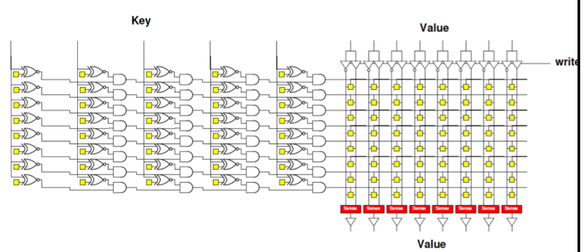


Assoc: 5b key, 8 entries, 8b value
How many memory bits?



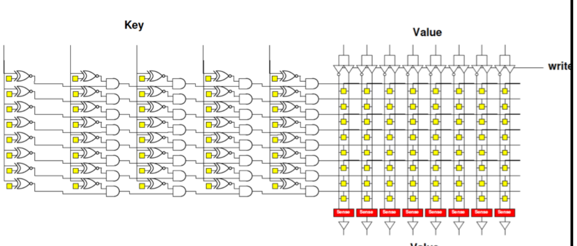
Direct: 8 entries, 8b value
How many memory bits?

Associative Memory Bank



- Memory cells = entries*(keybits+valuebits)

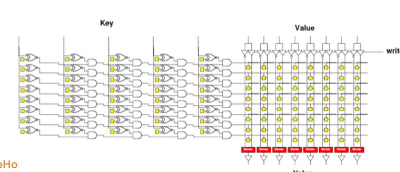
Associative Memory Bank



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

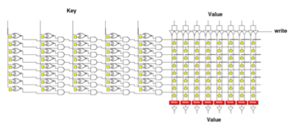
Associate Memory Cost

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



Associate Memory Cost

- Physical associative memory for 4KB LZW Chunk tree encode
  - 4K entries
  - 12b output
  - 12b+8b=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

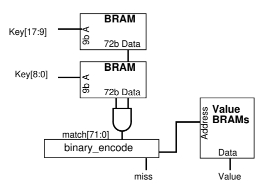
FPGA

- 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

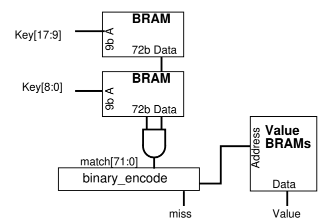
For wider match

- 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

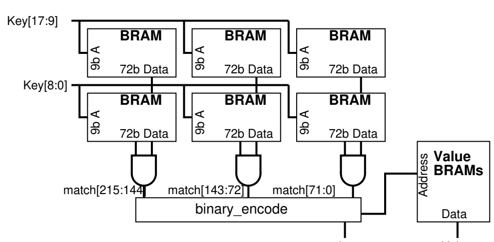


BRAM Associative Memory

- Previous slide expands match width
- How would we expand capacity?

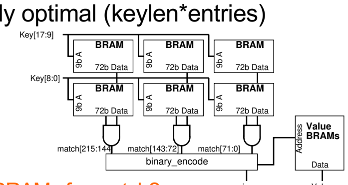


BRAM Associative Memory



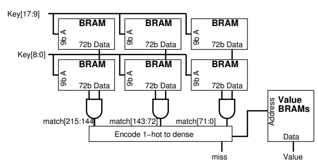
Associative Memory Cost

- 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?



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

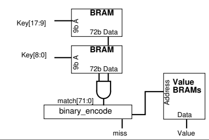
Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	1
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	0	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

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

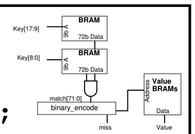
Value BRAM

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



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

Code Snippet



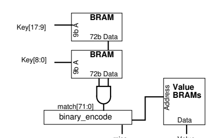

```
ap_uint<72> key_low[512];
ap_uint<72> key_high[512];
int value[72];

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];
```

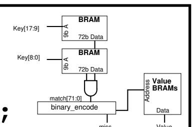

How Lookup Work?

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

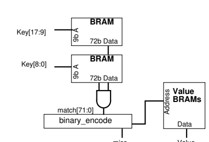
Addr	7	6	5	4	3	2	1	0
0x001	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	0	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

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

Value BRAM

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

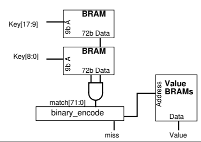


(only show bottom 8 b; rest 0's)

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

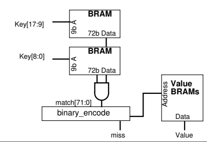
Addr	7	6	5	4	3	2	1	0
0x001	0	0	0	0	0	0	1	1
0x014	0	0	0	0	0	0	0	0
0x0C8	0	0	0	0	1	0	0	0
0x0F0	0	0	0	0	0	1	0	0
0x113	0	0	0	0	0	0	0	0

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)

Software Map

Part 3

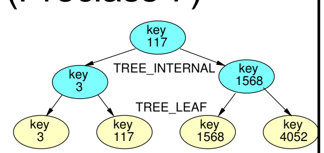
Software Map

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

Preclass 6

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

Tree Map (Preclass 7)



- 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

	BRAMs	Operations
Brute Search	1	4K
Tree with Dense RAM	512	1
Tree with Full Assoc	175	1
Tree with Tree	14	12
Tree with Hybrid (still to come)	52	1

36Kb BRAMs on ZU3EG = 216

Hash Tables

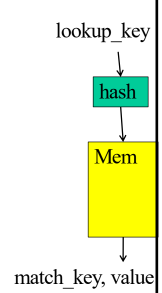
Part 4

High Performance Map

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

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



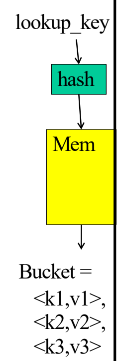
Miss = (match_key!=lookup_key)

Preclass 8a

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

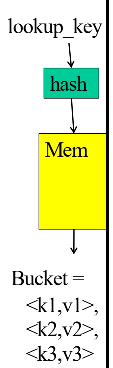
Hash Table

- 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



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



Preclass 8b

- Probability of conflict if N<<HASH_CAPACITY ?
  - Concrete example
    - N=4096
    - HASH_CAPACITY=409600
- Impact of HASH_CAPACITY on average bucket size?

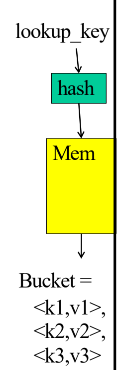
Hardware Hash Tables

Part 5

Skip to wrapup

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
- m – number of items in a slot
- Compute distribution for each bucket size

Preclass 9

N=1024

m→	0	1	2	3	4+
C=1024	0.37				
C=2048					
C=4096					

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

Preclass 9

N=1024

m→	0	1	2	3	4+
C=1024	0.37	0.37	0.18	0.061	0.019
C=2048	0.60	0.30	0.076	0.013	0.0017
C=4096	0.78	0.19	0.024	0.0020	0.00013

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

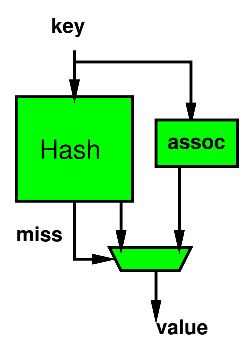
Hash

- 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

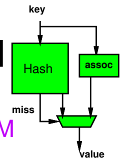
Idea

- 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 data 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 3
  - Prob 3+: <1% (see table 1024, 4096)
  - 20b key+12b value=4B per entry
  - 16384*3*4B=4*3*4 BRAMs
- 48+4=52 BRAMs

Further Optimization

- Previous example illustrative
  - Not necessarily optimal (explore parameters)
- 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

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
Brute Search	1	4K
Tree with Dense RAM	512	1
Tree with Full Assoc	175	1
Tree with Tree	14	12
Tree with Hybrid	52	1

36Kb BRAMs on ZU3EG = 216

Big Ideas

- Sparse, near O(1) Map access → Hash Table
- Rich design space for engineering associative map solutions

Admin

- Feedback (including HW6)
- Reading for Wednesday on Web
- First project milestone due Friday
  - Including teaming