ESE532:
System-on-a-Chip Architecture

Day 17: October 29, 2018

Associative Maps, Hash Tables

Today

- Motivation/Reminder
- LZW – tree
- Associative Memory
  - Custom
  - FPGA
- Software Maps
  - Hash Tables
- Hardware (FPGA) Hash Maps

Message

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

Reminder from Last Time

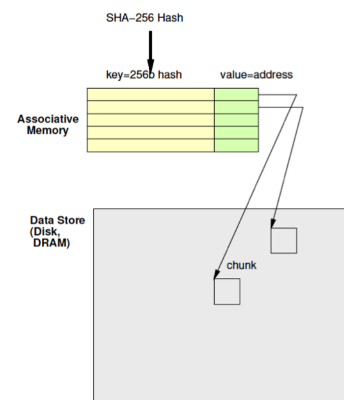
Day 16 Review

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

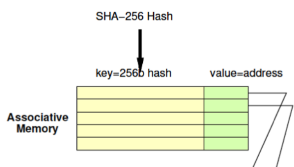
Deduplication Architecture



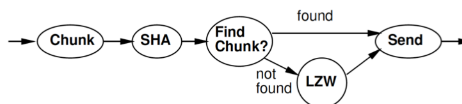
Day 16 Review

Associative Memory

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



LZW Compression



Day 16 Review

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 Review

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

Day 16 Review

Preclass 7

- DICT_SIZE comparisons per input character

```
#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);
}
```


Day 16 Review

Idea

- Avoid O(Dictionary-size) work
- Represent all strings as prefix tree
- Share prefix among substrings
- Follow prefix trees with fixed work per input character

Day 16 Review

Tree Algorithm

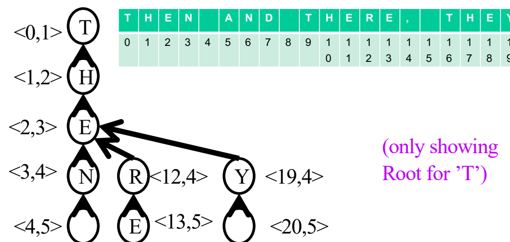
Root for each character

- 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

Day 16 Review

Tree Example

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



Day 16 Review

Large Memory Implementation

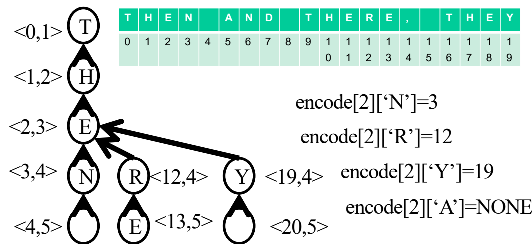
- int encode[SIZE][256];
- Name tree node by position in chunk
  - lastpos
- c is a character
- Encode[lastpos][c] holds the next tree node that extends tree node lastpos by c
  - Or NONE if there is no such tree node

Day 16 Review

Tree Example

(only showing Root for 'T')

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



Day 16 Review

Tree Algorithm

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

Memory Tree Algorithm

curr – pointer into input chunk

```
// follow tree
y=0; x=0;
while(encode[x][input[curr+y]]!=NONE)
    x=encode[x][input[curr+y]]; y++;
If (y>0) // send <name of last tree node>
  send <x,y>
else
  send input[curr+y]
encode[x][input[curr+y]]=curr+y // extend
```

Complexity

- How much work per character to encode?
  - When input character has a non-NONE match?
  - When input character finds a NONE?

Compact Memory

- int encode[SIZE][256];
- How many entries in this table are not NONE?
  - In processing a message of length SIZE, how many assignments are made to this array?

Compact Memory

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

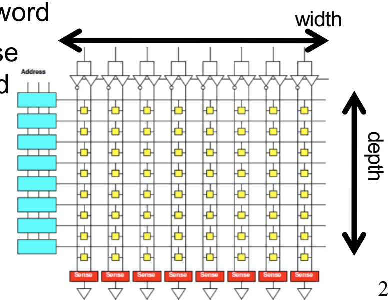
LZW So Far – 4KB chunks

- Brute Force
  - Needs one byte per byte = 4KB = 1 BRAM
  - DICT_SIZE=4096 comparisons per byte
- Dense memory encode[SIZE][256]
  - Need 2*256B byte = 512*4KB = 512 BRAMs
  - 1 comparison and lookup per byte

Custom Hardware Associative Memory

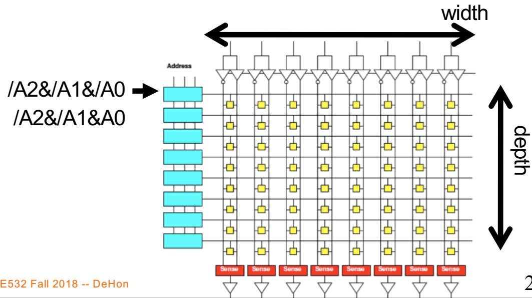
Memory Block Review

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

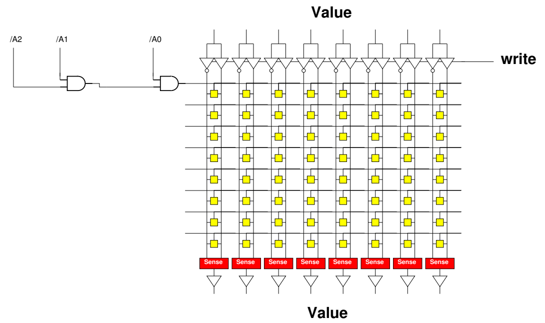


Address Blocks

- Each address match is AND

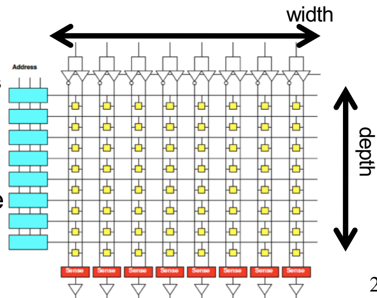


Address Blocks



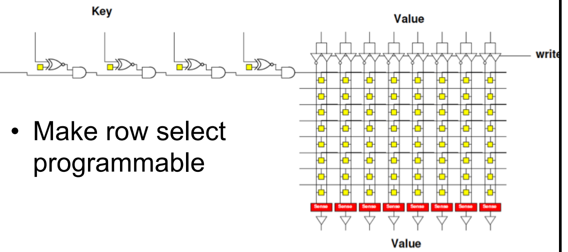
Memory Block Associative

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

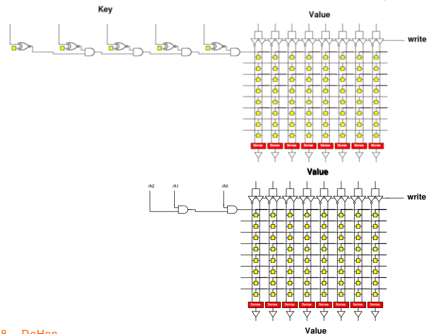


Programmable Key

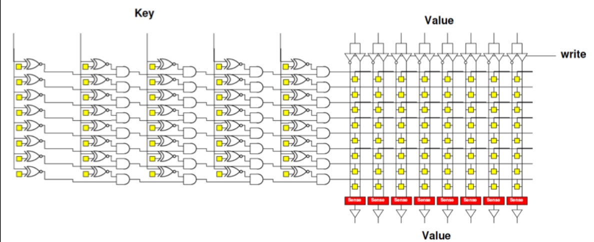
- Make row select programmable



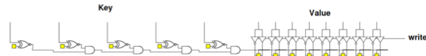
Contrast Assoc. and Dense Memory



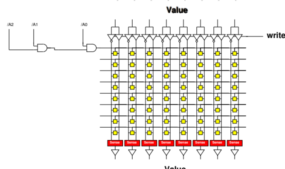
Associative Memory Bank



Programmable Key

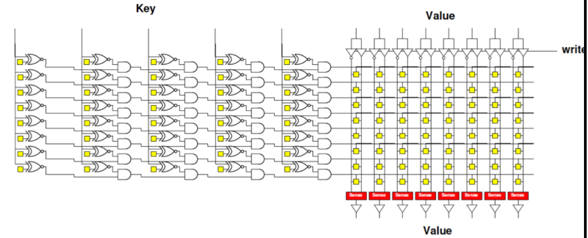


Assoc: 5b key, 8 entries, 8b value
How many memory bits?



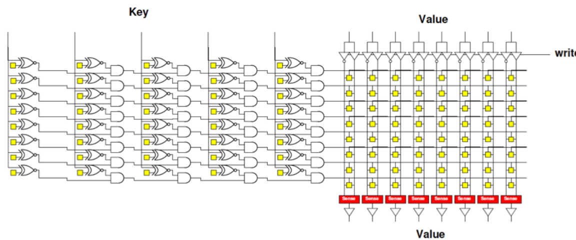
Direct: 8 entries, 8b value
How many memory bits?

Associative Memory Bank



- Memory cells = entries*(keybits+valuebits)

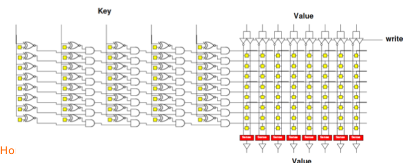
Associative Memory Bank



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

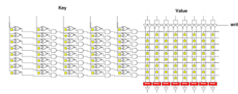
Associate Memory Cost

- More expensive than equal capacity SRAM memory bank
  - Memory cells in decoder
  - No sharing of AND-terms 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 4Kx8 memory (cells)?
- Compare 4096*256 with 2B result for dense LZW case (cells)?

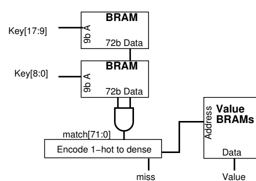
FPGA

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

Assoc. Mem from BRAM

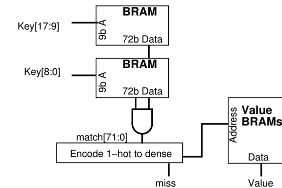
For wider match

- 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

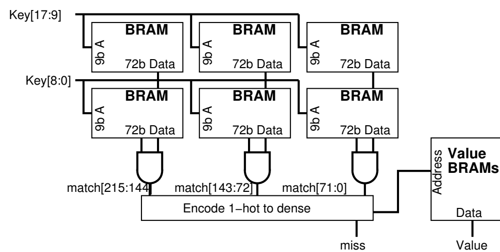


BRAM Associative Memory

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

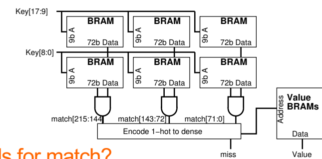


BRAM Associative Memory



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?



Software Map

Software Map

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

Preclass 1

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

Tree Map

- 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(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

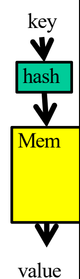
- Need to maintain balance
- Doable with O(log(N)) insert
  - Tricky
  - See Red-Black Tree

High Performance Map

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

Hash Table

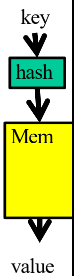
- Attempt to turn into direct lookup
- Compute some function of key
  - A hash
- Perform lookup at that point
- If hash maps a single entry (or no entry)
  - Great, got direct lookup
    - Like sparse table case



Preclass 3a

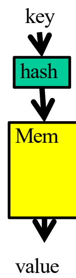
- Average number of entries per hash when N > HASH_CAPACITY?

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 3b

- Probability of conflict if N<<HASH_CAPACITY ?
- Given above, how can we reduce bucket sizes?

Preclass 4

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 4

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

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

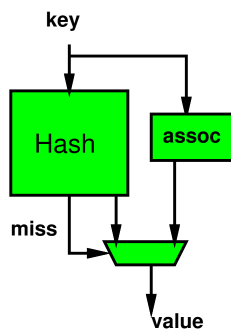
Hash

- 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

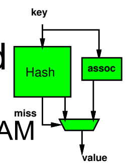
Idea

- 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



- With 3 match BRAMs + 1 data BRAM
  - Associative match 20b key
  - 72 entries (72/4096=1.7%)
- 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

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

Big Ideas

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

Admin

- No office hours Tuesday (tomorrow)
- First project milestone due Friday
  - Including teaming