

ESE532: System-on-a-Chip Architecture

Day 3: September 10, 2018
Memory



Today

Memory

- Memory Bottleneck
- Memory Scaling
- Latency Engineering
 - Scheduling
 - Data Reuse: Scratchpad and cache
- Bandwidth Engineering
 - Wide word
 - Banking (form of parallelism for data)

Message

- Memory bandwidth and latency can be bottlenecks
- Minimize data movement
- Exploit small, local memories
- Exploit data reuse

Memory Scaling

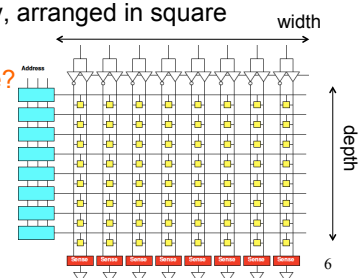
On-Chip Delay

- Delay is proportional to distance travelled
- Make a wire twice the length
 - Takes twice the latency to traverse
 - (can pipeline)
- Modern chips
 - Run at 100s of MHz to GHz
 - Take 10s of ns to cross the chip
 - Takes 100s of ns to reference off-chip data
- What does this say about placement of computations and memories?

Memory Block

- Linear wire delay with distance
- Assume constant area/memory bit
- N-bit memory, arranged in square

• 4x capacity
delay change?



Memory Block

- Linear wire delay with distance
- Assume constant area/memory bit
- N-bit memory, arranged in square
- Width relate to N?
- Depth?
- Delay scale N?

Penn ESE532 Fall 2018 -- DeHon

Memory Block Bandwidth

- N-bit memory, arranged in square
- Bits/cycle scale with N?

Penn ESE532 Fall 2018 -- DeHon

Memory Block Energy

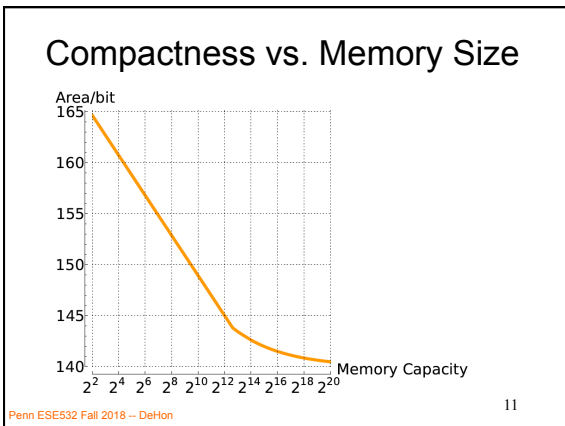
- Assume read full width,
 - Energy scales with N
 - Activate \sqrt{N} wires
 - Each wire \sqrt{N} long
 - Or, energy/bit scales as \sqrt{N}
- Larger memories cost more energy

Penn ESE532 Fall 2018 -- DeHon

Memory Block Density

- Address, sense amplifiers are large
- Scale slower than bit area
 - Bits scale N
 - Address $\sqrt{N} \cdot \log(N)$
 - Sense amps \sqrt{N}
- Large memories pay less per bit
 - denser

Penn ESE532 Fall 2018 -- DeHon



Memory Scaling

- Small memories are fast
 - Large memories are slow
- Small memories low energy
 - Large memories high energy
- Large memories dense
 - Small memories cost more area per bit
- Combining:
 - Dense memories are slow

Penn ESE532 Fall 2018 -- DeHon

Latency Engineering Scheduling

Penn ESE532 Fall 2018 -- DeHon

13

Preclass 2

- 10 cycle latency to memory
- Throughput in each case?

Case 1:

```
for(i=0;i<MAX;i++) {  
    in=a[i]; // memory read  
    out=f(in); // 10 cycle compute  
    b[i]=out;  
}
```

Case 2:

```
next_in=a[0];  
for(i=1;i<MAX;i++) {  
    in=next_in;  
    next_in=a[i];  
    out=f(in);  
    b[i-1]=out;  
}  
b[MAX-1]=f(next_in);
```

Penn ESE532 Fall 2018 -- DeHon

14

Lesson

- Long memory latency can impact throughput
 - When must wait on it
 - When part of a cyclic dependency
- Overlap memory access with computations when possible
 - exploit parallelism between compute and memory access

Penn ESE532 Fall 2018 -- DeHon

15

Latency Engineering Data Reuse

Penn ESE532 Fall 2018 -- DeHon

16

Preclass 3abc

- $MAX=10^6$ $WSIZE=5$
- How many times execute $t+=x[]*w[]$?
- How many reads to $x[]$ and $w[]$?

Runtime
read x, w 20
 $t+=...$ 5

```
for (i=0;i<MAX;i++) {  
    t=0;  
    for (j=0;j<WSIZE;j++)  
        t+=x[i+j]*w[j];  
    y[i]=t;  
}
```

Penn ESE532 Fall 2018 -- DeHon

17

Preclass 3c

- $T=MAX*WSIZE(T_{comp}+T_x+T_w)$
- $T=10^6*5*(5+20+20)$

$T=2.25*10^8$

```
for (i=0;i<MAX;i++) {  
    t=0;  
    for (j=0;j<WSIZE;j++)  
        t+=x[i+j]*w[j];  
    y[i]=t;  
}
```

Penn ESE532 Fall 2018 -- DeHon

18

Preclass 3d

- How many distinct x[], w[] read?

```

for (i=0; i<MAX; i++) {
    t=0;
    for (j=0; j<WSIZE; j++)
        t+=x[i+j]*w[j];
    y[i]=t;
}

```

Penn ESE532 Fall 2018 -- DeHon 9

Strategy

- Add small, local memory bank
 - Small memories are faster

Penn ESE532 Fall 2018 -- DeHon 20

Preclass 3e

- How use local memory to reduce large memory reads to 3d?
- How much local memory need?
- How much faster might small memory be?

```

for (i=0; i<MAX; i++) {
    t=0;
    for (j=0; j<WSIZE; j++)
        t+=x[i+j]*w[j];
    y[i]=t;
}

```

Penn ESE532 Fall 2018 -- DeHon

Preclass 3

```

for (i=0; i<MAX; i++) {
    t=0;
    for (j=0; j<WSIZE; j++)
        t+=x[i+j]*w[j];
    y[i]=t;
}

for (j=0; j<WSIZE; j++)
    local_w[j]=w[j];
for (j=0; j<WSIZE-1; j++)
    local_x[j+1]=x[j];
for (i=0; i<MAX; i++) {
    t=0;
    for (j=0; j<WSIZE-1; j++)
        local_x[j]=local_x[j+1];
    local_x[WSIZE-1]=x[i+WSIZE-1];
    for (j=0; j<WSIZE; j++)
        t+=local_x[j]*local_w[j];
    y[i]=t;
}

```

Penn ESE532 Fall 2018 -- DeHon --

Preclass 3

- $WSIZE * T_w$
- $WSIZE * T_x$
- $WSIZE * MAX * T_{local}$
- $MAX * T_x$
- $WSIZE * MAX * T_{comp}$
- $WSIZE * MAX * 2 * T_{local}$

```

for (j=0; j<WSIZE; j++)
    local_w[j]=w[j];
for (j=0; j<WSIZE-1; j++)
    local_x[j+1]=x[j];
for (i=0; i<MAX; i++) {
    t=0;
    for (j=0; j<WSIZE-1; j++)
        local_x[j]=local_x[j+1];
    local_x[WSIZE-1]=x[i+WSIZE-1];
    for (j=0; j<WSIZE; j++)
        t+=local_x[j]*local_w[j];
    y[i]=t;
}

```

Penn ESE532 Fall 2018 -- DeHon 23

Preclass 3

- $WSIZE * MAX * (T_{comp} + 3 * T_{local})$
- $+ WSIZE * (T_w + T_x)$
- $+ MAX * T_x$
- $5 * 10^6 * (5 + 3)$
- $+ 5 * (20 + 20)$
- $+ 10^6 * 20$
- $6 * 10^7$

```

for (j=0; j<WSIZE; j++)
    local_w[j]=w[j];
for (j=0; j<WSIZE-1; j++)
    local_x[j+1]=x[j];
for (i=0; i<MAX; i++) {
    t=0;
    for (j=0; j<WSIZE-1; j++)
        local_x[j]=local_x[j+1];
    local_x[WSIZE-1]=x[i+WSIZE-1];
    for (j=0; j<WSIZE; j++)
        t+=local_x[j]*local_w[j];
    y[i]=t;
}

```

Penn ESE532 Fall 2018 -- DeHon 24

Preclass 3

```

for (i=0; i<MAX; i++) {
    t=0;
    for (j=0; j<WSIZE; j++)
        t+=x[i+j]*w[j];
    y[i]=t;
}

for (j=0; j<WSIZE; j++)
    local_w[j]=w[j];
for (j=0; j<WSIZE-1; j++)
    local_x[j+1]=x[j];
for (i=0; i<MAX; i++) {
    t=0;
    for (j=0; j<WSIZE-1; j++)
        local_x[j]=local_x[j+1];
    local_x[WSIZE-1]=x[i+WSIZE-1];
    for (j=0; j<WSIZE; j++)
        t+=local_x[j]*local_w[j];
    y[i]=t;
}
    
```

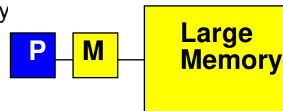
• $2.25 \cdot 10^8$ $6 \cdot 10^7$
 • $3.75x$

Lesson

- Data can often be reused
 - Keep data needed for computation in
 - Closer, smaller (faster, less energy) memories
 - Reduces latency costs
 - Reduces bandwidth required from large memories
- Reuse hint: value used multiple times
 - Or value produced/consumed

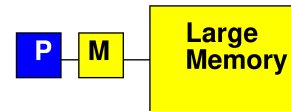
Processor Data Caches

- Traditional Processor Data Caches are a heuristic instance of this
 - Add a small memory local to the processor
 - It is fast, low latency
 - Store anything fetched from large/remote memory in local memory
 - Hoping for reuse in near future
 - On every fetch, check local memory before go to large memory



Cache

- **Goal:** performance of small memory with density of large memory



Processor Data Caches

- Demands more than a small memory
 - Need to sparsely store address/data mappings from large memory
 - Makes more area/delay/energy expensive than just a simple memory of capacity
- Don't need explicit data movement
- Cannot control when data moved/saved
 - Bad for determinism
- Limited ability to control what stays in small memory simultaneously

Terminology

- Cache
 - Hardware-managed small memory in front of larger memory
- Scratchpad
 - Small memory
 - Software (or logic) managed
 - Explicit reference to scratchpad vs. large (other) memories
 - Explicit movement of data

Terminology: Unroll Loop

- Replace loop with instantiations of body

```

For WSIZE=5
for (i=0;i<MAX;i++) {
  t=0;
  for (j=0;j<WSIZE;j++)
    t+=x[i+j]*w[j];
  y[i]=t;
}

for (i=0;i<MAX;i++) {
  t=0;
  t+=x[i+0]*w[0];
  t+=x[i+1]*w[1];
  t+=x[i+2]*w[2];
  t+=x[i+3]*w[3];
  t+=x[i+4]*w[4];
  y[i]=t;
}

```

Penn ESE532 Fall 2018 -- DeHon

31

Terminology: Unroll Loop

- Replace loop with instantiations of body

```

For WSIZE=5
for (i=0;i<MAX;i++) {
  t=0;
  for (j=0;j<WSIZE;j++)
    t+=x[i+j]*w[j];
  y[i]=t;
}

for (i=0;i<MAX;i++) {
  t=x[i+0]*w[0]+x[i+1]*w[1]+x[i+2]*w[2]
  +x[i+3]*w[3]+x[i+4]*w[4];
  y[i]=t;
}

```

Penn ESE532 Fall 2018 -- DeHon

32

Terminology: Unroll Loop

- Can unroll partially

```

For WSIZE even
for (i=0;i<MAX;i++) {
  t=0;
  for (j=0;j<WSIZE;j++)
    t+=x[i+j]*w[j];
  y[i]=t;
}

for (i=0;i<MAX;+=2) {
  t=0;
  for (j=0;j<WSIZE;j+=2) {
    t+=x[i+j]*w[j];
    t+=x[i+j+1]*w[j+1];
  }
  y[i]=t;
}

```

Penn ESE532 Fall 2018 -- DeHon

33

Amdahl's Law

Penn ESE532 Fall 2018 -- DeHon

34

Amdahl's Law

- If you only speedup Y(%) of the code, the most you can accelerate your application is $1/(1-Y)$
- $T_{\text{before}} = 1*Y + 1*(1-Y)$
- Speedup by factor of S
- $T_{\text{after}} = (1/S)*Y + 1*(1-Y)$
- Limit $S \rightarrow \text{infinity}$ $T_{\text{before}}/T_{\text{after}} = 1/(1-Y)$

Penn ESE532 Spring 2018 -- DeHon

35

Amdahl's Law

- $T_{\text{before}} = 1*Y + 1*(1-Y)$
- Speedup by factor of S
- $T_{\text{after}} = (1/S)*Y + 1*(1-Y)$
- $Y=70\%$
 - Possible speedup ($S \rightarrow \text{infinity}$) ?
 - Speedup if $S=10$?

Penn ESE532 Spring 2018 -- DeHon

36

Amdahl's Law

- If you only speedup $Y(\%)$ of the code, the most you can accelerate your application is $1/(1-Y)$
- Implications
 - Amdahl: good to have a fast sequential processor
 - Keep optimizing
 - $T_{\text{after}} = (1/S) * Y + 1 * (1-Y)$
 - For large S , bottleneck now in the $1-Y$

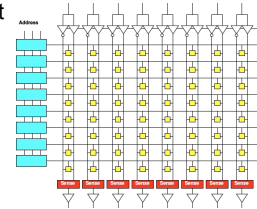
Bandwidth Engineering

Bandwidth Engineering

- High bandwidth is easier to engineer than low latency
 - Wide-word
 - Banking
 - Decompose memory into independent banks
 - Route requests to appropriate bank

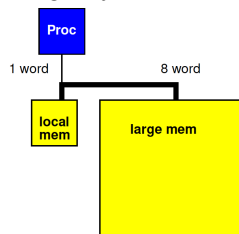
Wide Memory

- Relatively easy to have a wide memory
- As long as we share the address
 - One address to select wide word or bits
- Efficient if all read together



Revisit Preclass 3

- Use wide memory access to move 8 words ($x[]$) between large and small memory in a single cycle



Preclass 3 + wide

- Impact of 8 word datapath between large and local memory?

$$\bullet \text{ WSIZE} * \text{MAX} * (T_{\text{comp}} + 3 * T_{\text{local}})$$

$$\bullet + \text{ WSIZE} * (T_w + T_x)$$

$$\bullet + \text{ MAX} * T_x$$

$$\bullet \frac{5 * 10^6 * (5 + 3)}{5 * (20 + 20)}$$

$$\bullet + 10^6 * 20$$

$$\bullet 6 * 10^7$$

- What term most impacted?

```

for (j=0; j<WSIZE; j++)
    local_w[j]=w[j];
for (j=0; j<WSIZE-1; j++)
    local_x[j+1]=x[j];
for (i=0; i<MAX; i++) {
    t=0;
    for (j=0; j<WSIZE-1; j++)
        local_x[j]=local_x[j+1];
    local_x[WSIZE-1]=x[i+WSIZE-1];
    for (j=0; j<WSIZE; j++)
        t+=local_x[j]*local_w[j];
    y[i]=t;
}
    
```

Preclass 3 + wide

- $WSIZE * MAX * (T_{comp} + 3 * T_{local})$
- $+ WSIZE * (T_w + T_x)$
- $+ MAX / 8 * T_x$
- $+ MAX * (T_{local} + 2)$
- $5 * 10^6 * (5 + 3)$
- $+ 5 * (20 + 20)$
- $+ 10^6 * 20 / 8$
- $+ 10^6 * 3$
- $4.55 * 10^7$

```

for (j=0; j<WSIZE; j++)
    local_w[j]=w[j];
for (j=0; j<WSIZE-1; j++)
    local_x[j+1]=x[j];
offset=0;
for (i=0; i<MAX; i++) {
    t=0;
    for (j=0; j<WSIZE-1; j++)
        local_x[j]=local_x[j+1];
    if (offset==0)
        local_x[WSIZE-1-WSIZE+1+8]=
            x[i+WSIZE-1-i+WSIZE-1+8]; // 8 words
    else
        local_x[WSIZE-1]=local_x[WSIZE-1+offset];
    offset=(offset+1)&(0x07); // mod 8
    for (j=0; j<WSIZE; j++)
        t+=local_x[j]*local_w[j];
    y[i]=t;
}
    
```

Preclass 4

```

for (i=0; i<MAX; i++) {
    y[i]=a[i]*b[i];
}
for (i=0; i<MAX; i++) {
    y[i]=a[b[i]];
}
    
```

Cycles reading a, b from large memory.
20 cycle read latency; MAX=10⁶

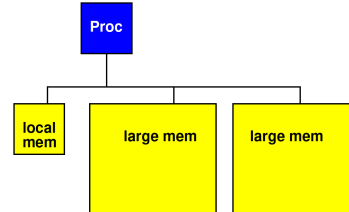
	a[]*b[]	a[b[]]
1-word wide		
8-word wide		

Lesson

- Cheaper to access wide/contiguous blocks memory
 - In hardware
 - From the architectures typically build
- Can achieve higher bandwidth on large block data transfer
 - Than random access of small data items

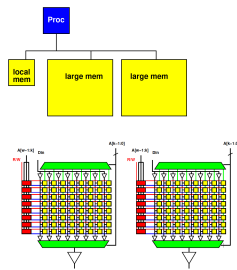
Bank Memory

- Break memory into independent banks
 - Allow banks to operate concurrently



Bank Memory

- Break memory into independent banks
 - Allow banks to operate concurrently



Preclass 4 Banked

```

for (i=0; i<MAX; i+=2) {
    local_a_even=a[i];
    local_a_odd=a[i+1];
    local_b_even=b[i];
    local_b_odd=b[i+1];
    y[i]=local_a_even*local_b_even;
    y[i+1]=local_a_odd*local_b_odd;
}
    
```

20 cycle read latency; MAX=10⁶
Put odd a, b in one bank, even in other.
Cycles reading a, b from large memory?

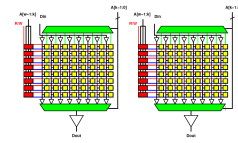
Preclass 4 Banked

```
for (i=0;i<MAX;i+=2) {
    local_b_even=b[i];
    local_b_odd=b[i+1];
for (i=0;i<MAX;i++) {
    y[i]=a[b[i]];
    local_a_even=a[local_b_even];
    local_a_odd=a[local_b_odd];
    y[i]=local_a_even;
    y[i+1]=local_a_odd;
}
```

20 cycle read latency; MAX=10⁶
Put odd a, b in one bank, even in other.
Cycles reading a, b from large memory?

Banking Costs

- Area compare
 - One memory bank
 - Depth 1024, width 64
 - Two memory banks
 - Depth 1024, width 32



Memory Organization

- Architecture contains
 - Large memories
 - For density, necessary sharing
 - Small memories local to compute
 - For high bandwidth, low latency, low energy
- Need to move data
 - Among memories
 - Large to small and back
 - Among small

Big Ideas

- Memory bandwidth and latency can be bottlenecks
- Exploit small, local memories
 - Easy bandwidth, low latency, energy
- Exploit data reuse
 - Keep in small memories
- Minimize data movement
 - Small, local memories keep distance short
 - Minimally move into small memories

Admin

- Reading for Wednesday on canvas
- HW2 due Friday