08 - Address Generator Unit (AGU)

Andreas Ehliar

September 30, 2013
Today's lecture

- Memory subsystem
- Address Generator Unit (AGU)
Memory subsystem

- Applications may need from kilobytes to gigabytes of memory
- Having large amounts of memory on-chip is expensive
- Accessing memory is costly in terms of power
- Designing the memory subsystem is one of the main challenges when designing for low silicon area and low power
Memory issues

- Memory speed increases slower than logic speed
- Impact on Memory size
  - Small size memory: Fast and area inefficient
  - Large size memory: Slow and area efficient
Comparison: Traditional SRAM vs ASIC memory block

- **Traditional memory**
  - Single tri-state data bus for read/write
  - *Asynchronous* operation

- **ASIC memory block**
  - Separate buses for data input and data output
  - *Synchronous* operation
Medium to large sized SRAM memories are almost always based on this architecture.

That is, it is not possible to have a large asynchronous memory!
Best practices for memory usage in RTL code

- Use synchronous memories for all but the smallest memory blocks
- Register the data output as soon as possible
  - A little combinational logic could be ok, but don’t put a multiplier unit here for example
- Some combinational logic before the inputs to the memory is ok
  - The physical layout of the chip can cause delays here that you don’t see in the RTL code
Best practices for memory usage in RTL code

- Disable the enable signal to the memory when you are not reading or writing
  - This will save you a lot of power
ASIC memories

- ASIC synthesis tools are usually not capable of creating optimized memories.
- Specialized memory compilers are used for this task.
- Conclusion: You can’t implement large memories in VHDL or Verilog, you need to instantiate them.
  - An inferred memory will be implemented using standard cells which is very inefficient (10x larger than an inferred memory and much slower).
Inferred vs instantiated memory

```verilog
reg [31:0] mem [511:0];

always @(posedge clk) begin
  if (enable) begin
    if (write) begin
      mem[addr] <= writedata;
    end else begin
      data <= mem[addr];
    end
  end
end
```
Memory design in a core

- Memory is not located in the core
  - Memory address generation is in the core
  - Memory interface is in the core
Scratch pad vs Cache memories

- Scratch pad memory
  - Simpler, cheaper, and use less power
  - It has more deterministic behavior
  - Suitable for embedded / DSP
  - May exist in separate address spaces
Scratch pad vs Cache memories

- Cache memory
  - Consumes more power
  - Cache miss induced cycles costs uncertainty
  - Suitable for general computing, general DSP
  - Global address space.
  - Hide complexity
## Selecting Scratch pad vs Cache memory

<table>
<thead>
<tr>
<th>Low addressing complexity</th>
<th>High addressing complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cache</strong></td>
<td>Good candidate when aiming for quick TTM</td>
</tr>
<tr>
<td>If you are lazy</td>
<td><strong>Scratch pad</strong></td>
</tr>
<tr>
<td>Good candidate when aiming for low power/cost</td>
<td>Difficult to implement and analyze</td>
</tr>
</tbody>
</table>
The cost of caches

- More silicon area (tag memory and memory area)
- More power (need to read both tag and memory area at the same time)
  - If low latency is desired, several ways in a set associative cache may be read simultaneously as well.
- Higher verification cost
  - Many potential corner cases when dealing with cache misses.
Regardless of whether cache or scratch pad memory is used, address generation must be efficient.

Key characteristic of most DSP applications:
- Addressing is deterministic.
Typical AGU location
Basic AGU functionality

[Input] → Address calculation logic circuit → Address pointer → Combinational output → Registered output → Keeper → Initial address → Addressing feedback

[Source: Liu2008]
AGU Example

Memory direct
A <= Immediate data

Address register + offset
A <= AR + Immediate data

Register indirect
A <= RF

Register + offset
A <= RF + immediate data

Addr. reg. post increment
A <= AR; AR <= AR + 1

Addr. reg. pre decrement
AR <= AR - 1; A <= AR;

Address register + register
A <= RF + AR
Modulo addressing

- Most general solution
  - Address = TOP + AR % BUFFERSIZE
  - Modulo operation too expensive in AGU
    - (Unless BUFFERSIZE is a power of two)
- More practical:
  - AR = AR + 1
  - If AR is more than BOT, correct by setting AR to TOP
Lets add Modulo addressing to the AGU:

- A=AR; AR = AR + 1
- if(AR == BOT) AR = TOP;
What about post-decrement mode?
The programmer can exchange TOP and BOTTOM.
- Alternative - Add hardware to select TOP and BOTTOM based on which instruction is used.
Sometimes it makes sense to use a larger stepsize than 1

- In this case we can’t check for equality but must check for greater than or less than conditions
- $\text{if}(\ AR \ > \ \text{BOTTOM}) \ AR \ = \ AR \ - \ \text{BUFFERSIZE}$
Variable step size

Keepers and registers for BUFFERSIZE, STEPSIZE, and BOTTOM not shown.

Note that STEPSIZE can’t be larger than BUFFERSIZE.
Bit reversed addressing

- Important for FFT:s and similar creatures
- Typical behavior from FFT-like transforms:

<table>
<thead>
<tr>
<th>Input sample</th>
<th>Transformed sample</th>
<th>Output index (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>X[0]</td>
<td>000</td>
</tr>
</tbody>
</table>
Bit reversed addressing

- Case 1: Buffer size and buffer location can be fixed
- Case 2: Buffer size is fixed, buffer location is arbitrary
- Case 3: Buffer size and location are not fixed
Case 1: Buffer size and buffer location can be fixed

Solution: If buffer size is $2^N$, place the start of the buffer at an even multiple of $2^N$

```
ADDR = {FIXED_PART, BIT_REVERSE(ADDR_COUNTER_N_BITS);}
```
Case 2: Buffer size is fixed, buffer location is arbitrary

Solution: If buffer size is $2^N$, place the start of the buffer at an even multiple of $2^N$

$$\text{ADDR} = \text{BASE\_REGISTER} + \text{BIT\_REVERSE(ADDR\_COUNTER\_N\_BITS)};$$
Bit reversed addressing

- Case 3: Buffer size and location are not fixed
- The most programmer friendly solution. Can be done in several ways.
Why design for memory hierarchy

- Small size memory is faster
  - Acting data / program in small size memories
- Large size memory is area efficient
  - Volume storage using large size memories
Typical SoC Memory Hierarchy

[Liu2008]
Memory partition

Requirements
- The number of data simultaneously
- Supporting access of different data types
- Memory shutting down for low power
- Overhead costs from memory peripheral
- Critical path from memory peripheral
- Limit of on chip memory size
Issues with off-chip memory

- Relatively low clockrate compared to on-chip
  - Will need clock domain crossing, possibly using asynchronous FIFOs
- High latency
  - Many clockcycles to send a command and receive a response
- Burst oriented
  - Theoretical bandwidth can only be reached by relatively large consecutive read/write transactions
Why burst reads from external memory?

- Procedure for reading from (DDR-)SDRAM memory
  - Read out one column (with around 2048 bits) from DRAM memory into flip-flops (slow)
  - Do a burst read from these flip-flops (fast)
  - Write back all bits into the DRAM memory

- Conclusion: If we have off-chip memory we should use burst reads if at all possible
Example: Image processing

- Typical organization of framebuffer memory
  - Linear addresses

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>160</td>
<td>161</td>
<td>162</td>
<td>163</td>
<td>164</td>
</tr>
<tr>
<td>1</td>
<td>320</td>
<td>321</td>
<td>322</td>
<td>323</td>
<td>324</td>
</tr>
<tr>
<td>2</td>
<td>480</td>
<td>481</td>
<td>482</td>
<td>483</td>
<td>484</td>
</tr>
<tr>
<td>3</td>
<td>640</td>
<td>641</td>
<td>642</td>
<td>643</td>
<td>644</td>
</tr>
</tbody>
</table>
Example: Image processing

- Fetching an 8x8 pixel block from main memory
  - Minimum transfer size: 16 pixels
  - Minimum alignment: 16 pixels
- Must load: 256 bytes
Rearranging memory content to save bandwidth

- Use tile based frame buffer instead of linear

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>640</td>
<td>641</td>
<td>642</td>
<td>643</td>
<td>656</td>
<td></td>
</tr>
</tbody>
</table>
Rearranging memory content to save bandwidth

• Fetching 8x8 block from memory
  – Minimum transfer size: 16 pixels
  – Minimum alignment: 16 pixels
• Must load: 144 pixels
  – Only 56% of the previous example!

Andreas Ehliar

08 - Address Generator Unit (AGU)
Rearranging memory content to save bandwidth

- Extra important when using a wide memory bus and/or a cache
Buses

- External memory
  - SDRAM, DDR-SDRAM, DDR2-SDRAM, etc
- SoC bus
  - AMBA, PLB, Wishbone, etc
  - You still need to worry about burst length, latency, etc
DMA definition and specification

- DMA: Direct memory access
  - An external device independent of the core
  - Running load and store in parallel with DSP
  - DSP processor can do other things in parallel

- Requirements
  - Large bandwidth and low latency
  - Flexible and support different access patterns
  - For DSP: Multiple access is not so important
Data memory access policies

- Both DMA and DSP can access the data memory simultaneously
  - Requires a dual-port memory
- DSP has priority, DMA must use spare cycles to access DSP memory
  - Verifying the DMA controller gets more complicated
- DMA has priority, can stall DSP processor
  - Verifying the processor gets more complicated
- Ping-pong buffer
  - Verifying the software gets more complicated
Ping-pong buffers
A simplified DMA architecture

Source decoding

In FIFO buffer

Destination encoding

Source port address counter

Down counter & FSM

Destination port address counter

Channel configuration

[Liu2008]
A typical DMA request

- Start address in data memory
- Start address in off-chip memory
- Length
- Priority
  - Permission to stall DSP core
  - Priority over other users of off-chip memory
- Interrupt flag
  - Should the DSP core get an interrupt when the DMA is finished?
Scatter Gather-DMA

- Pointer to DMA request 1
- Pointer to DMA request 2
- Pointer to DMA request 3
- End of List

- Start address in data memory
- Start address in off-chip memory
- Length
- Priority
  - Permission stall DSP core
  - Priority over other users of off-chip memory
- Interrupt flag
  - Should the DSP core get an interrupt when the DMA is finished?
Discussed on the whiteboard

- Why a cache takes more power than a scratch pad memory
- Schematic of a basic AGU including a control table
- Bit-reversed addressing in an AGU