Embedded Hardware (1)

Kai Huang
News: PS4 and Xbox One are Coming
# The Hardware

<table>
<thead>
<tr>
<th></th>
<th>PS4</th>
<th>Xbox One</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>semi-custom x86 AMD APU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>28nm 8-core Jaguar CPU</td>
<td></td>
</tr>
<tr>
<td>CPU frequency</td>
<td>1.6 GHz</td>
<td>1.75 GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>18 CUs: 1152 shaders (800MHz)</td>
<td>12 CUs: 768 shader (853 MHz)</td>
</tr>
<tr>
<td>Memory</td>
<td>8G 5500MHz DDR5</td>
<td>8G 2133MHz DDR3</td>
</tr>
<tr>
<td>Mem Bandwidth</td>
<td>176GB/sec</td>
<td>68.3GB/sec</td>
</tr>
<tr>
<td>Embedded SRAM</td>
<td>N/A</td>
<td>32MB (204GB/sec)</td>
</tr>
</tbody>
</table>
SoC Components

15 special purpose processors to offload CPU & GPU

- PCIe
- Audio Processors
- Audio DMA
- AV Out/Resize/Crop/ST
- AV In
- Video Encode
- Video Decode
- Swizzle/LZ Encode
- Swz LZ/MJPEG Dcd
- Swizzle Copy
- Swizzle Copy
- Graphics Core

Host Guest IO MMU

CPU

CPU-Cache-Coherent Memory Access

Any DRAM data can be coherent with CPU caches

DRAM CNTR

68 GB/sec peak BW
4 x 64 bit

2 GB DDR3

2 GB DDR3

2 GB DDR3

30 GB/sec coherent BW

CPU, GPU, special processors, and IO share memory via host-guest MMUs and synchronized page tables

109 GB/sec min
204 GB/sec peak BW
4 x 256 bit read & write

8 MB 8 MB 8 MB 8 MB
Dataflow MoC Recap

Kahn Process Network

Synchronous DataFlow

\[ h \text{ init} = 1 \]
\[ h \text{ init} = 0 \]

\[ g \]

\[ f \]

\[ h2 \]

\[ g \]

\[ f \]

\[ h1 \]
Outline

- Processor
- Memory
- I/O
Outline

- Processor
  - Single-cycle datapath
  - Pipeline datapath
  - Processor types
- Memory
- I/O
Y-Chart Methodology

Architecture model

Mapping

Performance Evaluation

Performance Numbers

Applications model

CPU

Mem

CPU Bus

C1, C2

Bridge

IP Bus

P1 P3

P2 P4

P5

Arbiter

HW

P1 d P3

C1

C2

P1 d P2

P2 d P4

C2

C1

P3

P4

P5
Embedded System Hardware

Embedded system hardware is frequently used in a loop ("hardware in a loop"): 

This course

A/D converter  
Sample-and-hold

Information processing

Display

D/A converter

Sensors

Environment

Actuators

Embedded system
Since 1946 all computers have had 5 components

Control unit coordinates various actions:
• Input,
• Output
• Processing

Datapath: the part of the central processing unit (CPU) that does the actual computations

Input unit accepts information:
• Human operators,
• Electromechanical devices
• Other computers

Output unit sends results of processing:
• To a monitor display,
• To a printer

Stores information:
• Instructions,
• Data
Datapath Components

- Combinational Elements
  - ALU, Adder
  - Immediate extender
  - Multiplexers

- Storage Elements
  - Instruction memory
  - Data memory
  - PC register
  - Register file

- Clocking methodology
  - Timing of reads and writes
1. ALU is a digital circuit that performs Arithmetic (Add, Sub, . . .) and Logical (AND, OR, NOT) operations.

2. John Von Neumann proposed the ALU in 1945 when he was working on EDVAC.
Multifunction ALU

Shift Operation
- None = 00
- SLL = 01
- SRL = 10
- SRA = 11

Arithmetic Operation
- ADD = 0
- SUB = 1

Logical Operation
- AND = 00
- OR = 01
- NOR = 10
- XOR = 11

Shift = 00
SLT = 01
Arith = 10
Logic = 11

SLT: ALU does a SUB and check the sign and overflow
Single-Cycle Datapath (with Control Signal)
Register Transfer Level (RTL)

- RTL is a description of data flow between registers
- RTL gives a meaning to the instructions
- All instructions are fetched from memory at address PC

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RTL Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Reg(Rd) ← Reg(Rs) + Reg(Rt);</td>
<td>PC ← PC + 4</td>
</tr>
<tr>
<td>SUB</td>
<td>Reg(Rd) ← Reg(Rs) – Reg(Rt);</td>
<td>PC ← PC + 4</td>
</tr>
<tr>
<td>ORI</td>
<td>Reg(Rt) ← Reg(Rs)</td>
<td>zero_ext(Im16);</td>
</tr>
<tr>
<td>LW</td>
<td>Reg(Rt) ← MEM[Reg(Rs) + sign_ext(Im16)];</td>
<td>PC ← PC + 4</td>
</tr>
<tr>
<td>SW</td>
<td>MEM[Reg(Rs) + sign_ext(Im16)] ← Reg(Rt);</td>
<td>PC ← PC + 4</td>
</tr>
</tbody>
</table>
| BEQ         | if (Reg(Rs) == Reg(Rt))  
            |    PC ← PC + 4 + 4 × sign_extend(Im16)  
            | else  PC ← PC + 4 |
Instructions are Executed in Steps

- **R-type**  
  - Fetch instruction: \( \text{Instruction} \leftarrow \text{MEM}[\text{PC}] \)  
  - Fetch operands: \( \text{data1} \leftarrow \text{Reg}(\text{Rs}), \text{data2} \leftarrow \text{Reg}(\text{Rt}) \)  
  - Execute operation: \( \text{ALU\_result} \leftarrow \text{func}(\text{data1}, \text{data2}) \)  
  - Write ALU result: \( \text{Reg}(\text{Rd}) \leftarrow \text{ALU\_result} \)  
  - Next PC address: \( \text{PC} \leftarrow \text{PC} + 4 \)

- **I-type**  
  - Fetch instruction: \( \text{Instruction} \leftarrow \text{MEM}[\text{PC}] \)  
  - Fetch operands: \( \text{data1} \leftarrow \text{Reg}(\text{Rs}), \text{data2} \leftarrow \text{Extend}(\text{imm16}) \)  
  - Execute operation: \( \text{ALU\_result} \leftarrow \text{op}(\text{data1}, \text{data2}) \)  
  - Write ALU result: \( \text{Reg}(\text{Rt}) \leftarrow \text{ALU\_result} \)  
  - Next PC address: \( \text{PC} \leftarrow \text{PC} + 4 \)

- **BEQ**  
  - Fetch instruction: \( \text{Instruction} \leftarrow \text{MEM}[\text{PC}] \)  
  - Fetch operands: \( \text{data1} \leftarrow \text{Reg}(\text{Rs}), \text{data2} \leftarrow \text{Reg}(\text{Rt}) \)  
  - Equality: \( \text{zero} \leftarrow \text{subtract}(\text{data1}, \text{data2}) \)  
  - Branch: \( \text{if (zero)} \) \( \text{PC} \leftarrow \text{PC} + 4 + 4 \times \text{sign\_ext}(\text{imm16}) \)  
    \( \text{else} \) \( \text{PC} \leftarrow \text{PC} + 4 \)
### Instruction Execution Examples

- **LW**  
  **Lw Rt,C(Rs)**  
  - Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}]$  
  - Fetch base register: $\text{base} \leftarrow \text{Reg}(\text{Rs})$  
  - Calculate address: $\text{address} \leftarrow \text{base} + \text{sign\_extend}(\text{imm16})$  
  - Read memory: $\text{data} \leftarrow \text{MEM}[\text{address}]$  
  - Write register Rt: $\text{Reg}(\text{Rt}) \leftarrow \text{data}$  
  - Next PC address: $\text{PC} \leftarrow \text{PC} + 4$

- **SW**  
  **Sw Rt,C(Rs)**  
  - Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}]$  
  - Fetch registers: $\text{base} \leftarrow \text{Reg}(\text{Rs}), \text{data} \leftarrow \text{Reg}(\text{Rt})$  
  - Calculate address: $\text{address} \leftarrow \text{base} + \text{sign\_extend}(\text{imm16})$  
  - Write memory: $\text{MEM}[\text{address}] \leftarrow \text{data}$  
  - Next PC address: $\text{PC} \leftarrow \text{PC} + 4$

- **Jump**  
  **j C**  
  - Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}]$  
  - Target PC address: $\text{target} \leftarrow \text{PC}[31:28], \text{Imm26}, \text{‘00’}$  
  - Jump: $\text{PC} \leftarrow \text{target}$
Execution of Load Instruction: \texttt{lw} Rt, C(Rs)

- ExtOp = ‘sign’ to sign-extend Immmediate16 to 32 bits
- RegDst = ‘0’ selects Rt as destination register
- ALUSrc = ‘1’ selects extended immediate as second ALU input
- ALUCtrl = ‘ADD’ to calculate data memory address as Reg(Rs) + sign-extend(Imm16)
- MemRead = ‘1’ to read data memory
- MemtoReg = ‘1’ places the data read from memory on BusW
- MemWrite = 0
- RegWrite = ‘1’ to write the memory data on BusW to register Rt
Execution of Store Instruction: \texttt{sw Rt,C(Rs)}

ExtOp = ‘sign’ to sign-extend Immediate16 to 32 bits

RegDst = ‘x’ because no destination register

ALUSrc = ‘1’ to select the extended immediate as second ALU input

ALUCtrl = ‘ADD’ to calculate data memory address as Reg(Rs) + sign-extend(Imm16)

MemWrite = ‘1’ to write data memory

MemtoReg = ‘x’ because we don’t care what data is placed on BusW

RegWrite = ‘0’ because no register is written by the store instruction
Execution of Jump Instruction: \textit{j C}

\textbf{J = 1 selects Imm26 as jump target address}

\textbf{Upper 4 bits are from the incremented PC}

\textbf{PCSrc = 1 to select jump target address}

\textbf{MemRead, MemWrite & RegWrite are 0}

\textbf{We don’t care about RegDst, ExtOp, ALUSrc, ALUCtrl, and MemtoReg}
Drawbacks of Single Cycle Processor

- Long cycle time
  - All instructions take as much time as the slowest

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Instruction Fetch</th>
<th>Reg Read</th>
<th>ALU</th>
<th>Reg Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Instruction Fetch</td>
<td>Reg Read</td>
<td>ALU</td>
<td>Memory Read</td>
</tr>
<tr>
<td>Store</td>
<td>Instruction Fetch</td>
<td>Reg Read</td>
<td>ALU</td>
<td>Memory Write</td>
</tr>
<tr>
<td>Branch</td>
<td>Instruction Fetch</td>
<td>Reg Read</td>
<td>ALU</td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>Instruction Fetch</td>
<td>Decoded</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Alternative Solution: Multicycle implementation
  - Break down instruction execution into multiple cycles
Single-Cycle vs. Multicycle

Clock

<table>
<thead>
<tr>
<th>Time needed</th>
<th>Time allotted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr 1</td>
<td>Instr 2</td>
</tr>
<tr>
<td>3 cycles</td>
<td>5 cycles</td>
</tr>
<tr>
<td>Instr 3</td>
<td>Instr 4</td>
</tr>
<tr>
<td>3 cycles</td>
<td>4 cycles</td>
</tr>
</tbody>
</table>

Time saved

Instr 1 Instr 4 Instr 3 Instr 2

Time needed

Instr 1 Instr 2 Instr 3 Instr 4

Time allotted

3 cycles 5 cycles 3 cycles 4 cycles
Outline

- Processor
  - Single-cycle datapath
  - Pipeline datapath
  - Processor types

- Memory

- I/O
Single-Cycle Datapath

- Shown below is the single-cycle datapath
- How to pipeline this single-cycle datapath?

**Answer:** Introduce registers at the end of each stage
Pipelined Datapath

- Pipeline registers, in green, separate each pipeline stage.
- Pipeline registers are labeled by the stages they separate.
- Is there a problem with the register destination address?
- Destination register number should come from MEM/WB
  - Along with the data during the written back stage
- Destination register number is passed from ID to WB stage
Graphically Representing Pipelines

- Multiple instruction execution over multiple clock cycles
  - Instructions are listed in execution order from top to bottom
  - Clock cycles move from left to right
  - Figure shows the use of resources at each stage and each cycle

Program Execution Order:
- **lw** $6, 8($5)
- **add** $1, $2, $3
- **ori** $4, $3, 7
- **sub** $5, $2, $3
- **sw** $2, 10($3)

Time (in cycles): CC1 → CC2 → CC3 → CC4 → CC5 → CC6 → CC7 → CC8

11/19/2013  kai.huang@tum
Instruction–Time Diagram

- Diagram shows:
  - Which instruction occupies what stage at each clock cycle
- Instruction execution is pipelined over the 5 stages

Up to five instructions can be in execution during a single cycle

ALU instructions skip the MEM stage. Store instructions skip the WB stage

<table>
<thead>
<tr>
<th>Instruction Order</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $7, 8($3)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>lw $6, 8($5)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>ori $4, $3, 7</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td>WB</td>
</tr>
<tr>
<td>sub $5, $2, $3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td>WB</td>
</tr>
<tr>
<td>sw $2, 10($3)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
</tbody>
</table>
Consider a 5-stage instruction execution in which ...
- Instruction fetch = ALU operation = Data memory access = 200 ps
- Register read = register write = 150 ps

What is the single-cycle non-pipelined time?
What is the pipelined cycle time?
What is the speedup factor for pipelined execution?

Solution
Non-pipelined cycle = 200+150+200+200+150 = 900 ps
Pipelined cycle time = max(200, 150) = 200 ps

CPI for pipelined execution = 1
- One instruction completes each cycle (ignoring pipeline fill)

Speedup of pipelined execution = 900 ps / 200 ps = 4.5
- Instruction count and CPI are equal in both cases

Speedup factor is less than 5 (number of pipeline stage)
- Because the pipeline stages are not balanced
## Summary between Datapaths

<table>
<thead>
<tr>
<th></th>
<th>Single Cycle</th>
<th>Multiple Cycle</th>
<th>Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clock Cycle Time</strong></td>
<td>Long (Long enough for the slowest instruction)</td>
<td>Short (long enough for the slowest instruction step)</td>
<td>Short (long enough for the slowest pipeline stage)</td>
</tr>
<tr>
<td><strong>Cycle Per Instruction</strong></td>
<td>1 clock cycle per instruction (by definition)</td>
<td>Variable number of clock cycles per instruction</td>
<td>Fixed number of clock cycles per instruction, one for each pipeline stage</td>
</tr>
<tr>
<td><strong># instruction executing concurrently</strong></td>
<td>1</td>
<td>1</td>
<td># pipeline stage</td>
</tr>
<tr>
<td><strong>Duplicate Hardware</strong></td>
<td>Yes, since we can use a functional unit (FU) for at most one subtask per instruction</td>
<td>No, since the instruction generally is broken into single-FU steps</td>
<td>Yes, to avoid restriction on pipeline execution</td>
</tr>
<tr>
<td><strong>Extra Register</strong></td>
<td>No</td>
<td>Yes, to hold results for the next step</td>
<td>Yes, to provide results for the pipeline stage</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>Baseline</td>
<td>Faster, but not too fast</td>
<td>Fastest, if pipeline is balanced</td>
</tr>
</tbody>
</table>

**11/19/2013**
kai.huang@tum
Outline

- Processor
  - Single-cycle datapath
  - Pipeline datapath
  - Processor types
- Memory
- I/O
General Purpose Processors (GPP)

- High performance
  - Highly optimized circuits and technology
  - Use of parallelism
    - superscalar: dynamic scheduling of instructions
    - super-pipelining: instruction pipelining, branch prediction, speculation
  - complex memory hierarchy
- Not suited for real-time applications
  - Execution times are highly unpredictable because of intensive resource sharing and dynamic decisions
- Properties
  - Good average performance for large application mix
  - High power consumption
GPP + Memory (I): von Neumann Architecture

1. PC := 200
2. Fetch => IR := Mem[PC]
3. Decode IR
4. Execute
5. PC := PC + 1
6. goto 2

ADD a1,a2,aN

Memory
Data + Program

CPU

PC

IR

GPR
GPP + Memory (II): Harvard Architecture

Program Memory

Data Memory

CPU

PC

IR

GPR

address

data

address

data
Intel Pentium 4 Northwood

Buffer Allocation & Register Rename
- Instruction Queue (for less critical fields of the uOps)
- General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)
- Floating Point, MMX, SSE, SSE2
- Renamed Register File 128 entries of 128 bits

uOp Schedulers
- FP Move Scheduler: (8x8 dependency matrix)
- Parallel (Matrix) Scheduler: for the two double pumped ALUs
- General Floating Point and Slow Integer Scheduler: (8x8 dependency matrix)
- Load / Store uOp Scheduler: (8x8 dependency matrix)
- Load / Store Linear Address Collision History Table

Integer Execution Core
1. uOp Dispatch unit & Replay Buffer Dispatches up to 6 uOps / cycle
2. integer Renamed Register File: 128 entries of 32 bit, 6 status flags (2 read ports and six write ports)
3. Data bus switch & Bypasses to and from the integer Register File
4. Flags, Write back
5. Double Pumped ALU
6. Double Pumped ALU
7. Load Address Generator Unit
8. Store Address Generator Unit
9. Load Buffer (48 entries)
10. Store Buffer (24 entries)
11. ROB Reorder Buffer 2x42 entries
12. 8kByte Level 1 Data cache four way set associative 1R/1W
13. Summed Address Index decode and Way Predict
14. Cache Line Read / Write Transferbuffers and 256 bit wide bus to and from L2 cache

Execution Pipeline Start
- Register Alias History Tables (2x128)
- Register Alias Tables
- uOp Queue

Instruction Trace Cache
- Micro code Sequencer
- Micro code ROM & Flash
- Trace Cache Fill Buffers
- Distributed Tag comparators 24 bit virtual Tags

Instruction Decoder
- Up to 4 decoded uOps/cycle (on (from max. one x86 instr/cycle)
- Instructions with more than four are handled by Micro Sequencer
- Trace Cache LRU bin
- Raw Instruction Bytes
- Data 14B, 64 entry fully associative, between threads dual ported (for loads and stores)

Instruction Fetch
- from L2 cache and Branch Prediction
- Front End Branch Prediction Tables (BTB), shared, 4/96 entries in total
- Instruction TLBs 2x4/entry, fully associative for 4k and 4M pages: 1x [31:12] Out: Physical address [35:12] + 2 page level bits

Trace Cache Access, next Address Predict
- Trace Cache Branch Prediction Table (BTB), 512 entries
- Return Stacks (2x16 entries)
- Trace Cache next IP's (2x)
- Miscellaneous Tag Data

Front Side Bus Interface, 400-800 MHz

April 19, 2003
www.chip-architect.com

11/19/2013 kai.huang@tum
Simple GPP: Xilinx MicroBlaze

IOPB: Instruction side On chip Peripheral Bus
IXCL_M: Instruction-side Xilinx Cache Link Master
IXCL_S: Instruction-side Xilinx Cache Link Slave
ILMB: Instruction side Local Memory Bus

DOPB: Data side On chip Peripheral Bus
DXCL_M: Data-side Xilinx Cache Link Master
DXCL_S: Data-side Xilinx Cache Link Slave
DLMB: Data side Local Memory Bus
MFSL: Master Fast Simplex Link
SFSL: Slave Fast Simplex Link
**Embedded Processors – RISC vs. CISC**

- **Complex instruction set CISC (e.g. x86)**
  - Map complexity of common instructions directly in machine code
  - Complex instructions can consist of several simple instructions
  - Can lead to subtle timing issues
  - Used in general purpose computing

- **Reduced instruction set RISC (e.g. ARM – Acorn Risc Machine)**
  - Only simple machine instructions; Compiler has to map high-level language onto simple instructions
  - All instructions take the same time
  - Used in embedded systems (Real-time hardware, smart phones, ...)

11/19/2013 kai.huang@tum
Application Specific Instruction Set Processors

- **Micro Controllers (MicroCtrl)**
  - Used in Control Dominated Systems
  - Reactive systems with event driven behavior
  - Application examples: cars, consumer electronics (washing machines, dishwashers etc.)

- **Digital Signal Processors (DSPs)**
  - Used in Data Dominated Systems
  - Streaming-oriented systems with mostly periodic behavior
  - Application examples: signal processing

- **Very Long Instruction Word Processors (VLIWs)**
  - Used in Data Dominated Systems
  - Application examples: image processing
ASIP: Micro Controllers

- Control-dominant applications
  - Supports process scheduling and synchronization
  - Preemption (interrupt), context switch
  - Short latency times
- Low power consumption
- Peripheral units often integrated
- Suited for real-time applications

### Philips 83 C552:
8 bit-8051 based microcontroller
ASIP: Digital Signal Processors

- Optimized for data-flow applications
- Suited for simple control flow
- Parallel hardware units
- Specialized instruction set
- High data throughput
- Zero-overhead loops
- Specialized memory

- Suited for real-time applications
Very Long Instruction Word Processors

- Key idea: detection of possible parallelism to be done by compiler, not by hardware at run-time (inefficient).
- VLIW: parallel operations (instructions) encoded in one long word (instruction packet), each instruction controlling one functional unit.
- VLIW processors are an example of the so called Explicit Parallelism Instruction Computers (EPIC)
- 5 issue slots (functional units FU),
- therefore up to 5 instructions can be executed in parallel
Application Specific Integrated Circuits (ASICs)

- Custom-designed circuits necessary
  - if ultimate speed or
  - energy efficiency is the goal and
  - large numbers can be sold.

- Approach suffers from
  - long design times,
  - lack of flexibility (changing standards)
  - high costs, i.e., Millions of $ mask costs
Reconfigurable Processing Units (RPUs)

- Full custom chips (HW) may be too expensive, software (SW) too slow.
- Combine the speed of HW with the flexibility of SW
  - HW with programmable functions and interconnect.
  - HW (Re-)Configurable at design-time or at run-time (dynamic reconfiguration)

Field Programmable Gate Arrays (FPGAs)
- Currently the most sophisticated and used RPUs
- Applications
  - Fast and very cheap prototyping of (MP-)SoCs
  - Encryption,
  - Fast “object recognition“ (medical and military)
  - Adapting mobile phones to different standards

- Very popular devices from
  - XILINX (Virtex II(Pro), Virtex 4, Virtex 5, Virtex 6, Virtex 7)
  - Altera (Cyclone, Arria, Stratix)
  - Actel and others
Floor-plan of VIRTEX II FPGAs

- Configurable Logic Block (CLB)
- Digital Clock Manager (DCM)
- Input/Output Blocks (IOB)
System-on-Chip (SoC)

SoC Components

- 15 special purpose processors offload CPU & GPU
- PCIe
- Audio Processors
- Audio DMA
- AV Out, Rsz Cmprst
- AV In
- Video Encode
- Video Decode
- Swizzle/LZ Encode
- Swz LZ/MJPG Dcd
- Swizzle Copy
- Swizzle Copy
- Graphics Core

CPU

CPU-Cache-Coherent Memory Access

Host Guest IO MMU

Host Guest GPU MMU

DRAM CNTR

68 GB/sec peak BW
4 x 64 bit

2 GB DDR3
2 GB DDR3
2 GB DDR3
2 GB DDR3

30 GB/sec coherent BW

109 GB/sec min
204 GB/sec peak BW
4 x 256 bit read & write

8 MB 8 MB 8 MB 8 MB

CPU, GPU, special processors, and IO share memory via host-guest MMUs and synchronized page tables

Any DRAM data can be coherent with CPU caches
The main difference between general purpose highest volume microprocessors and embedded systems is specialization.

Specialization should respect flexibility
- application domain specific systems shall cover a class of applications
- some flexibility is required to account for late changes, debugging

System analysis required
- identification of application properties which can be used for specialization
- quantification of individual specialization effects
Why Implementation Alternatives?

Trade-off between Flexibility and Performance/Power Efficiency
Energy Efficiency

"inherent power efficiency of silicon"

© Hugo De Man, IMEC, Philips, 2007