Performance Analysis of Embedded Systems

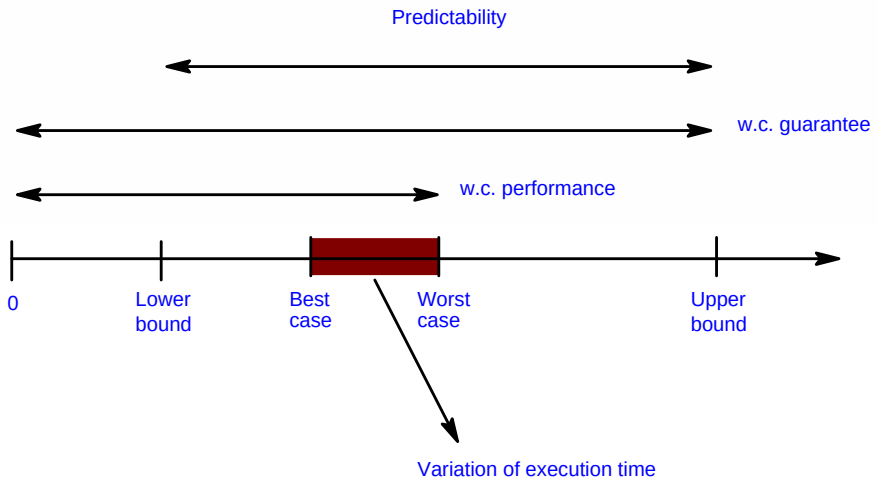
EE 382V – SoC
Fall 2009, UT Austin

- Introduction
- Algorithm analysis
- System-level exploration
- System-level performance estimation
  - Scheduling
- Analysis case studies

Performance of a System

- Depends on many factors
- System design (algorithms and data structures)
- Implementation (code)
- The workload to which it is subjected
- The metric used in the evaluation
- Interactions between these factors

Timing Analysis – Concepts



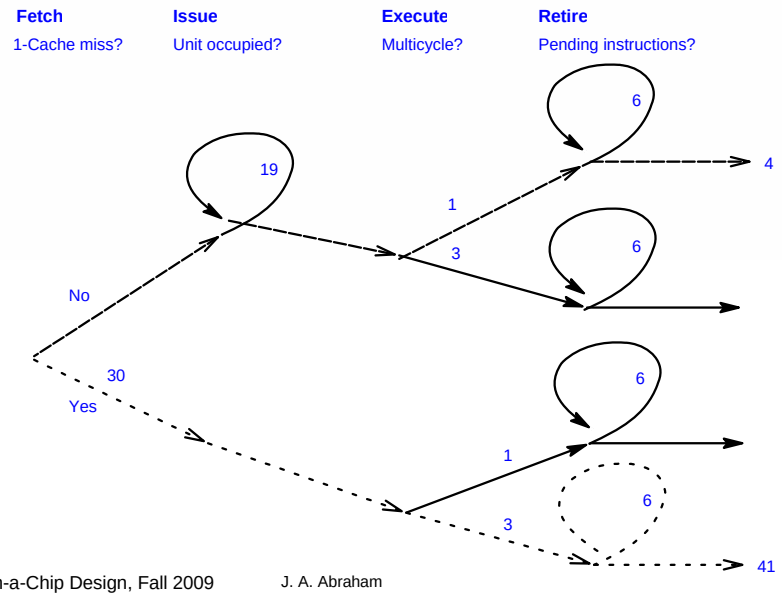
Evaluating a Design

- Algorithm
  - Analysis of complexity
  - Identify bottlenecks, evaluate tradeoffs
- Software partition
  - Analysis of code
  - Profiling execution for performance, power
  - Alternative implementations
- Hardware partition
  - Delay, performance estimation
  - Power estimation
  - Explore alternatives

Algorithm Analysis

- Example: Sorting
  - "Bubble" sort
  - Merge sort
- Example: Fourier transform
  - Discrete Fourier transform
  - Fast Fourier transform

Execution of Multiply Instruction



Analysis of Code

- Example: BCH encoding
- Code in C
- Find the number of XOR and AND operations performed in the loop as a function of *k*
- Assume *length* is 1024, and in any bit position, 0 and 1 are equally likely

Example Code

```
encode_bch()
{
        register int    i, j;
        register int    feedback;
        for (i = 0; i < length - k; i++)
                bb[i] = 0;
        for (i = k - 1; i >= 0; i--) {
                feedback = data[i] ^ bb[length - k - 1];
                if (feedback != 0) {
                        for (j = length - k - 1; j > 0; j--)
                                if (g[j] != 0)
                                        bb[j] = bb[j - 1] ^ feedback;
                                else
                                        bb[j] = bb[j - 1];
                        bb[0] = g[0] && feedback;
                } else {
                        for (j = length - k - 1; j > 0; j--)
                                bb[j] = bb[j - 1];
                        bb[0] = 0;
                }
        }
}
```

Profiling Code

- Include effect of processor instruction set and architecture
- Many profiling tools for data gathering and analysis
  - *gprof*, etc
- Various interfaces, levels of automation, and approaches to information presentation
- A lot of work in the high performance computing community

Improving Performance Analysis

- Better benchmarks
  - Need to reflect user applications
  - Related to actual code
- Performance monitoring tools
- Performance modeling and analysis
- Software tools to automatically or semi-automatically optimize user codes

Instrumentation Techniques

- Program Instrumentation Techniques
  - Manual : Programmer inserted directives
  - Automatic : No direct user involvement
    - Binary Rewriting
    - Dynamic Instrumentation
- Processor Instrumentation Techniques
  - Information includes timers, memory system performance, processor usage, etc.
  - Available mostly through special registers or memory mapped location.
    - Example: Pentium Pro provides performance data through MSRs (model (machine)-specific registers). These registers include 64 bit cycle clock and counts of memory read/write, L1 cache misses, pipeline flushes, etc.
  - Hardware assisted trace generation.

Instrumentation Techniques, Cont'd

- Operating System Instrumentation Techniques
  - Information includes behavior of virtual memory, file system, file cache etc.
  - Instrumentation in the form of APIs for applications to access these variables.
- Network Instrumentation Techniques
  - Ways of measuring
    - Passive
      - Example: RMON protocol defines SNMP MIB variables to report traffic statistics over hubs and switches.
    - Active
      - Example: Ping, NWS in grid style computing.

DRM Performance Measurement

- The DRM code was originally designed to run in a Linux environment, but in this course, we are going to make it run on an embedded processor (the ARM926EJS)
- Thus, the original code should be profiled in both Linux and ARM environments to measure performance and identify bottlenecks
- *gprof* is used to profile the DRM code in the Linux environment, and the *ARM profiler* is used to profile the code in the ARM environment

DRM Performance Measurement - Linux

- *gprof* allows you to learn where your program spends its time and which functions call other functions while it was executing
- There are two options to display the results
  - *flat profile* : total amount of time program spends executing each function
  - *call graph* : how much time was spent in each function and its child functions

DRM Performance Measurement - Linux

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
62.50	0.10	0.10	1526	0.07	0.10	CChannelEstimation::ProcessDataInternal(CParameter&)
25.00	0.14	0.04	1507	0.03	0.04	CChannelEstimation::UpdateWienerFiltCoef(double, double, double)
12.50	0.16	0.02	31647	0.00	0.00	CChannelEstimation::FreqOptimalFilter(int, int, double, double, double, int)
0.00	0.16	0.00	1482	0.00	0.00	CChannelEstimation::GetSigma(double&)
0.00	0.16	0.00	1482	0.00	0.00	CChannelEstimation::GetSNREstdB(double&) const
0.00	0.16	0.00	1482	0.00	0.00	CChannelEstimation::GetDelay() const
0.00	0.16	0.00	1428	0.00	0.00	CChannelEstimation::CalAndBoundSNR(double, double)
0.00	0.16	0.00	2	0.00	0.04	CChannelEstimation::InitInternal(CParameter&)
0.00	0.16	0.00	1	0.00	0.00	__static_initialization_and_destruction_0(int, int)

- This example shows the *flat profile* display of the *Channel Estimation* module in DRM
- We can see that three functions consumed most of the CPU time while the *ProcessDataInternal* function consumes more than half of the CPU time

DRM Performance Measurement - Linux

index	% time	self	children	called	name
		0.10	0.37	1526/1526	Module<std::complex<double>, CEquSig>::ProcessDataThreadSave(CParameter&)
[8]	10.4	0.10	0.37	1526	CChannelEstimation::ProcessDataInternal(CParameter&) [8]
		0.05	0.11	1505/1505	CTimeSyncTrack::Process(CParameter&, CMatlibVector<std::complex<double> >&, int, double&, double&) [18]
		0.04	0.10	1505/1507	CChannelEstimation::UpdateWienerFiltCoef(double, double, double) [19]
		0.07	0.00	1525/1525	CTimeWiener::Estimate(CVectorEx<std::complex<double> >*, CMatlibVector<std::complex<double> >&, CVector<int>&, CVector<std::complex<double> >&, double) [30]
		0.01	0.00	1505/11814	CShiftRegister<double>::AddEnd(double) [37]
		0.00	0.00	1505/204936	CMatlibVector<double>::Init(int, double) [55]
		0.00	0.00	1428/1428	CChannelEstimation::CalAndBoundSNR(double, double) [388]

- This example shows the *call graph* of the *ProcessDataInternal* function defined in the *Channel Estimation* module
- From this information, we can find functions that may not have used much time, but called other functions that did use unusual amounts of time
- For example, the *ProcessDataInternal* function spent more time in its children than in the function itself

DRM Performance Measurement - ARM

- The ARM profiler, *armprof*, displays an execution profile of a program from a profile data file generated by an ARM debugger (*armsd*)
- Profiling data is collected by *armsd* while the code is being executed
- The profiler can display a *flat profile* giving the percentage of time spent in each function

DRM Performance Measurement - ARM

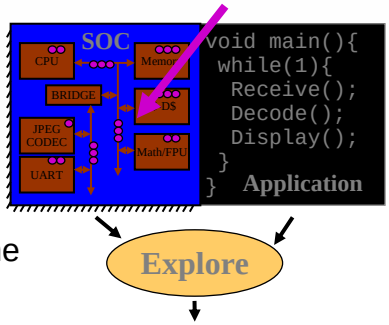
```
Name                    time%
ProcessDataInternal__18CChannelEstimationFR10CParameter
                        0.35%
-------------------------------------------------------------------
UpdateWienerFiltCoef__18CChannelEstimationFdN21
                        0.11%
-------------------------------------------------------------------
FreqOptimalFilter__18CChannelEstimationFiT1dN23T1
                        0.07%
-------------------------------------------------------------------
InitInternal__18CChannelEstimationFR10CParameter
                        0.00%
-------------------------------------------------------------------
GetSigma__18CChannelEstimationFRd
                        0.00%
-------------------------------------------------------------------
GetDelay__18CChannelEstimationCFv
                        0.00%
-------------------------------------------------------------------
_dmul                   28.40%
-------------------------------------------------------------------
_dadd                   12.91%
-------------------------------------------------------------------
```

- This example shows the flat profile result of DRM code (only functions related to *Channel Estimation* module are displayed here)
- Arm profiler displays both DRM functions (displayed in black) and ARM built-in functions (displayed in blue)
- Most of the ARM built-in functions are floating point emulation functions (and they consume a large portion of the ARM CPU time!)

System-Level Exploration

- Size = {1K, 4K, 8K}
- Line = {4, 8, 16}
- Assoc = {1, 2, 4}

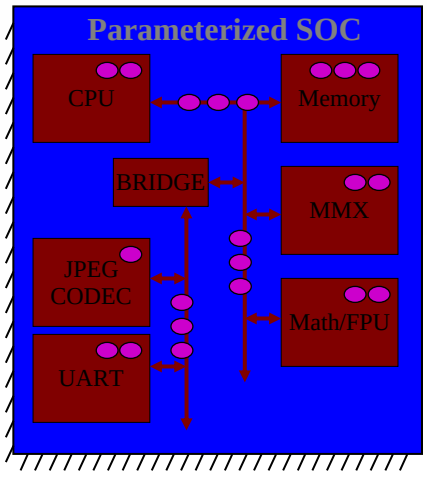
- Given:
  - Parameterized SOC architecture
  - Fixed application
- Automatically explore the design space
- Find optimal points w/respect to power and performance



Source: T. Givargis U.C.I.

Motivation

- Composed of 100s of cores
- Cores are "configurable"
- Configurations impact power/performance
- Large number of total configurations!

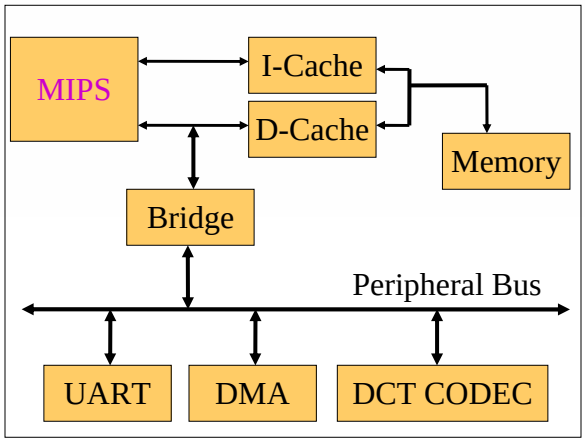


Architecture is otherwise fixed!

Source: T. Givargis U.C.I.

Target Architecture

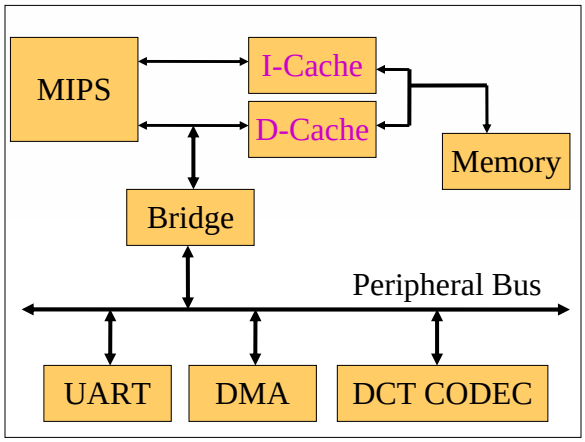
- Voltage scale
- Size, line, associativity
- Bus width, encoding (gray, invert)
- UART tx/rx buffer size
- DCT resolution



Source: T. Givargis
U.C.I.
Performance Analysis of Embedded Systems

Target Architecture

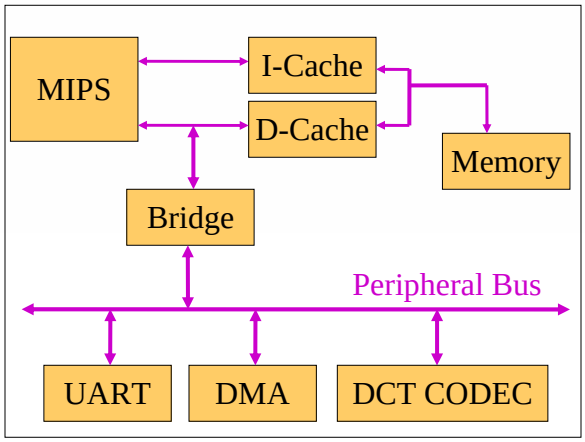
- Voltage scale
- Size, line, associativity
- Bus width, encoding (gray, invert)
- UART tx/rx buffer size
- DCT resolution



Source: T. Givargis
U.C.I.
Performance Analysis of Embedded Systems

Target Architecture

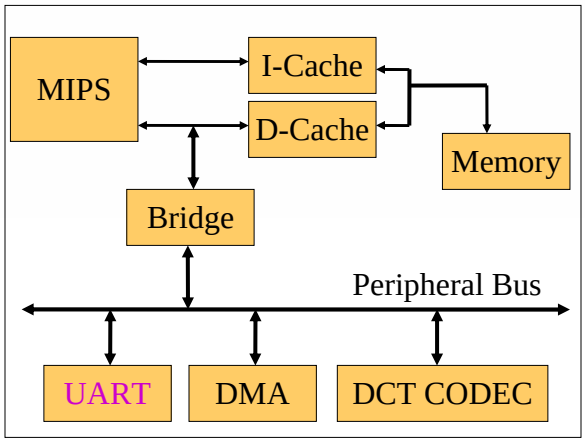
- Voltage scale
- Size, line, associativity
- Bus width, encoding (gray, invert)
- UART tx/rx buffer size
- DCT resolution



Source: T. Givargis U.C.I.

Target Architecture

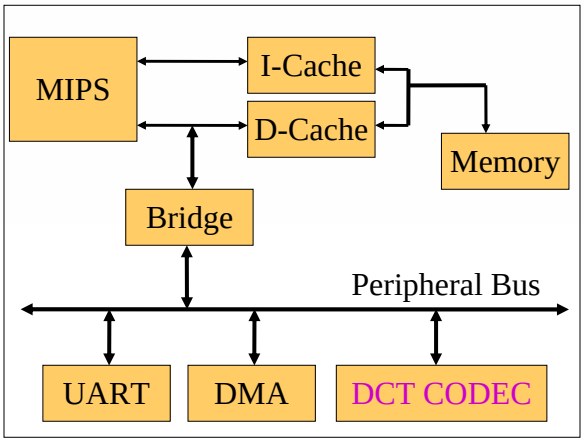
- Voltage scale
- Size, line, associativity
- Bus width, encoding (gray, invert)
- UART tx/rx buffer size
- DCT resolution



Source: T. Givargis U.C.I.

Target Architecture

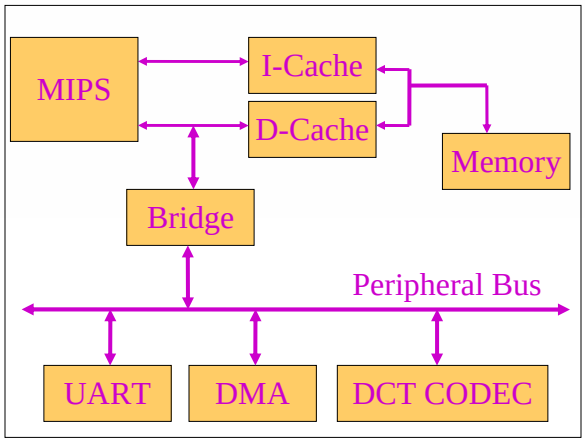
- Voltage scale
- Size, line, associativity
- Bus width, encoding (gray, invert)
- UART tx/rx buffer size
- DCT resolution



Source: T. Givargis U.C.I.

Target Architecture

- 26 parameters
- 10^{14} configurations
- What are the optimal configuration (given a fixed application)?



Source: T. Givargis U.C.I.

Exploration

Algorithm Idea

- A and B interdependent
- A and C are independent
- C and B are independent
- With knowledge about dependency we prune 98.6%
- Directed graph

A (10) * B (32) = 320 points

A (10) + C (32) = 42 points

C (32) + B (32) = 64 points

138 points

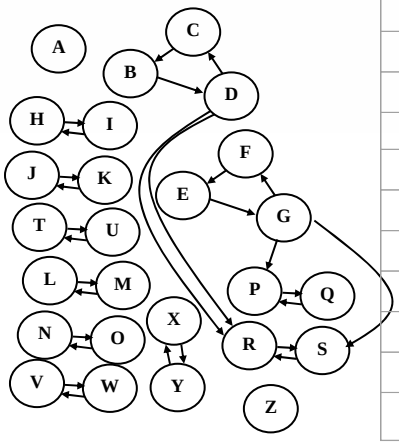
A (10) * B (32) * C (32) = 10240 points

Source: T. Givargis U.C.I.

Exploration

Source: T. Givargis U.C.I.

Dependency Graph

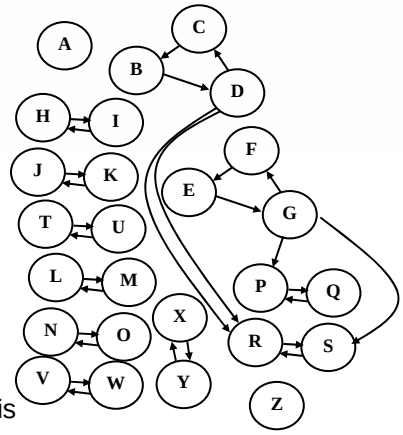


Node	Core	Parameter	Node	Core	Parameter
A	MIPS	Voltage scale	L	CPU D$ bus	Data bus width
B	I$	Total size	M		Data bus code
C		Line size	N		Addr bus width
D		Associativity	O		Addr bus code
E	D$	Total size	P	I/D$ Mem bus	Data bus width
F		Line size	Q		Data bus code
G		Associativity	R		Addr bus width
H	CPU I$ bus	Data bus width	S		Addr bus code
I		Data bus code	T	Peripheral bus	Data bus width
J		Addr bus width	U		Data bus code
K		Addr bus code	V		Addr bus width
X	UART	Tx buffer size	W		Addr bus code
Y		Rx buffer size	Z	DCT CODEC	Pixel resolution

Exploration

Dependency graph

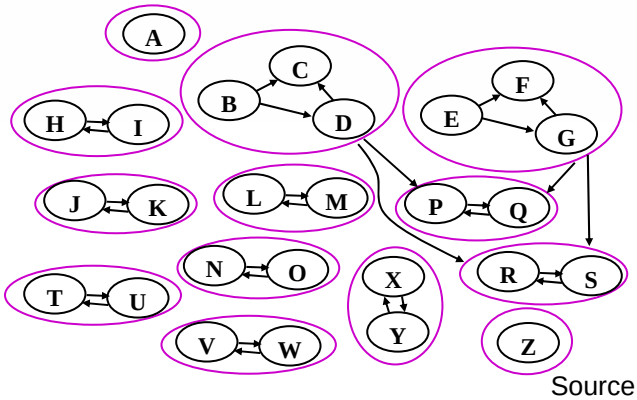
- Based on designer knowledge
- Computed by simulating all pairs of nodes *(quadratic time complexity, approx.)*
- One time effort



Source: T. Givargis U.C.I.

Exploration – Algorithm

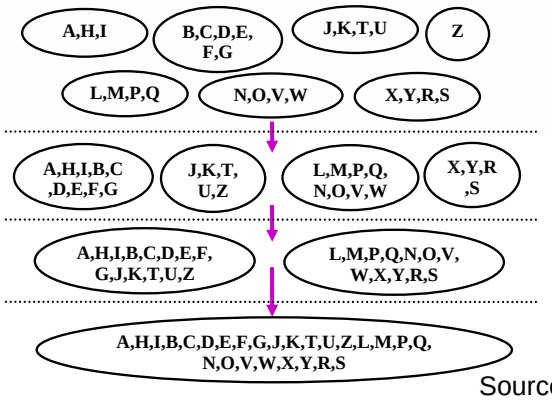
Step 1: Clustering followed by simulation



Source: T. Givargis U.C.I.

Exploration – Algorithm

Step 2: Pair-wise merge followed by simulation



Source: T. Givargis U.C.I.

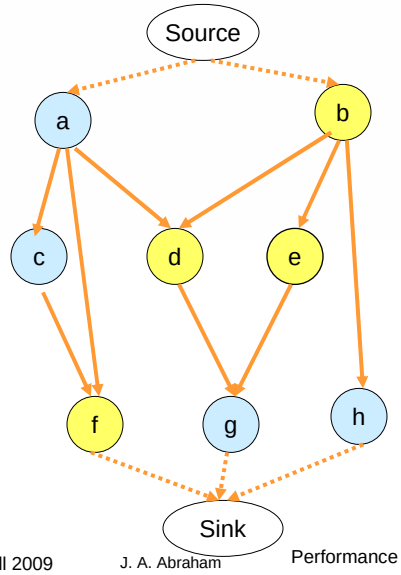
System Level Performance Estimation

- ASSP programming
  - Which process is mapped to what architectural resource?
- Structured ASIC/Generic SoC
  - What architectural building blocks are part of the system?
  - Which functionality is refined as a custom or IP hardware block?
  - What software processor are the code segments mapped to?
- Design decisions require evaluation of system-level performance

Task Graph Model

- A graph representation of the application specification.
- Derived from data dependency based representation commonly utilized in compilers
- Application is specified by a graph G(V,E)
  - V is the set of tasks
    - t(v,r) gives the run-time of “v” on a processing element “r”
  - E is the set of directed edges
    - e(u,v) implies data produced by u is consumed by v
    - v cannot begin execution before u has finished execution
- Multimedia and Network processing applications can be specified by this model

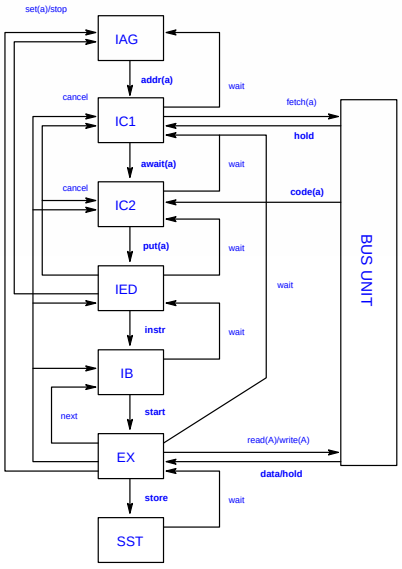
Task Graph Model



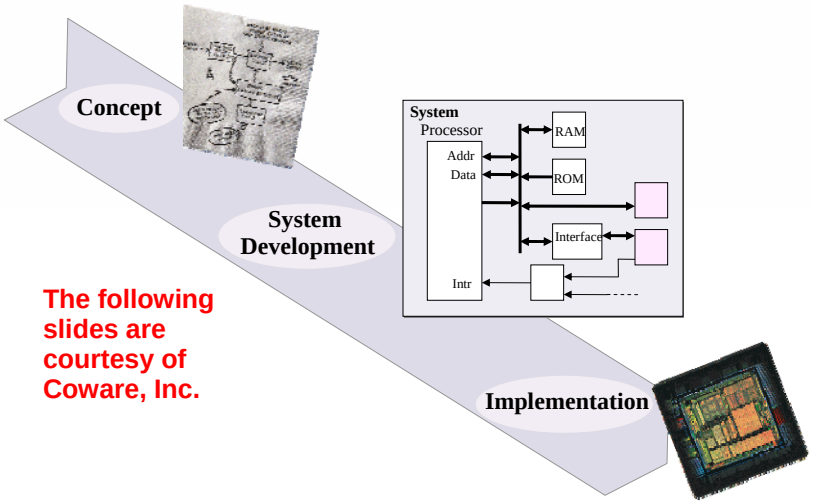
Analysis of Pipelined Processors

- Throughput versus latency
- Pipeline introduces dependencies on inputs (instructions)
- Hazards
  - Structural (resource being used by another)
  - Data (dependence for data calculations)
  - Control (calculating next address – branches, interrupts
- Look at abstract finite state machine (FSM) of pipeline
  - Try to decompose FSM
  - States may communicate with each other through *signals* (which may also carry data)

Abstract Pipeline Model of Motorola ColdFire 5307 Processor

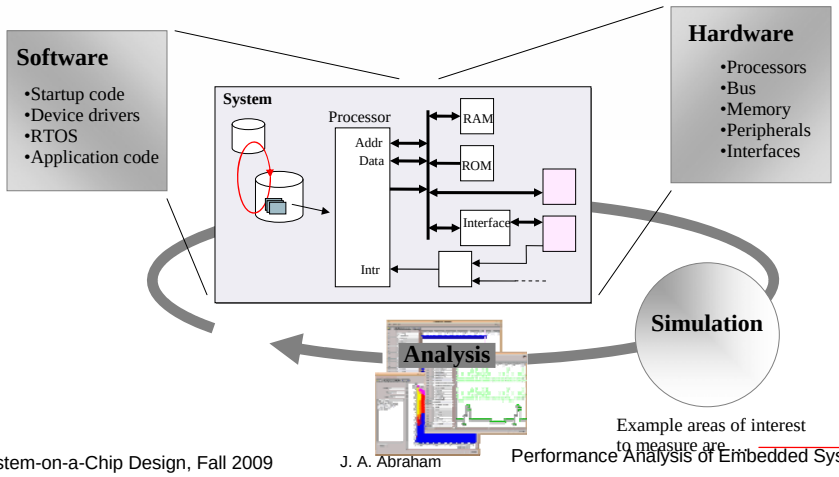


The purpose of analysis is to get the architecture right before implementation begins – the best design for system performance and functionality

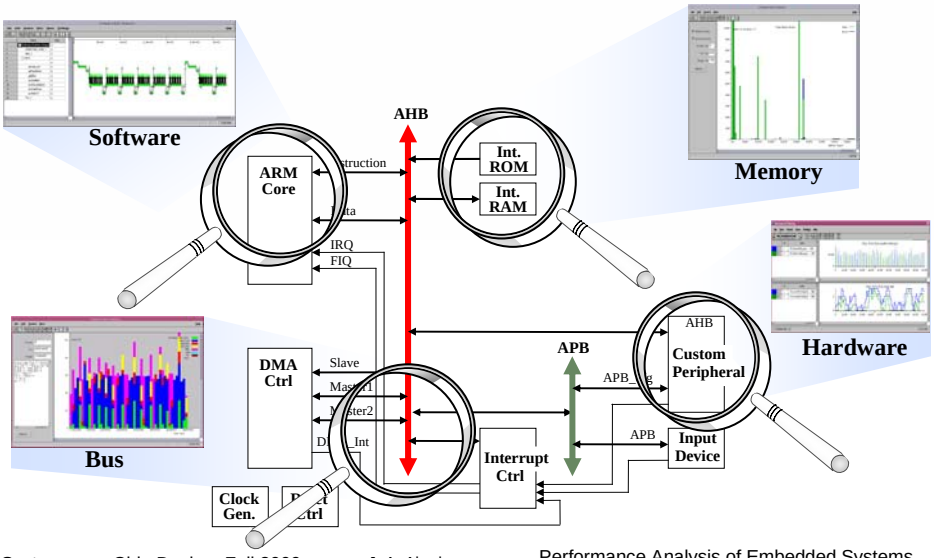


The following slides are courtesy of Coware, Inc.

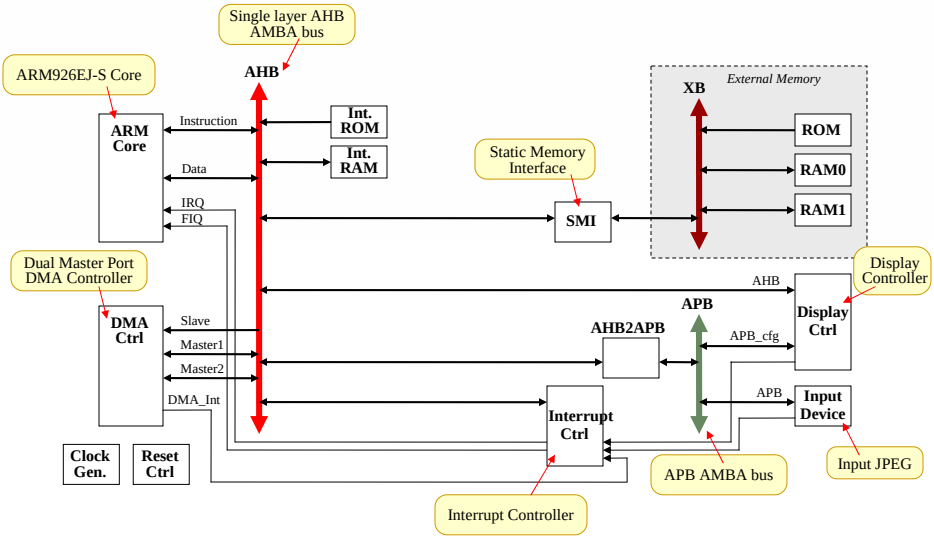
System development consists of putting together a system from a collection of hardware and software components…then iteratively measuring and modifying the system for optimum performance



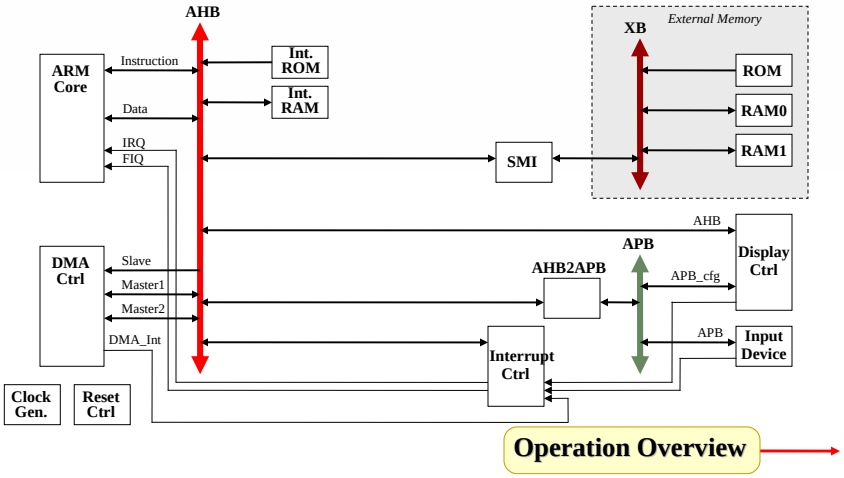
Areas of Interest for Analysis are:



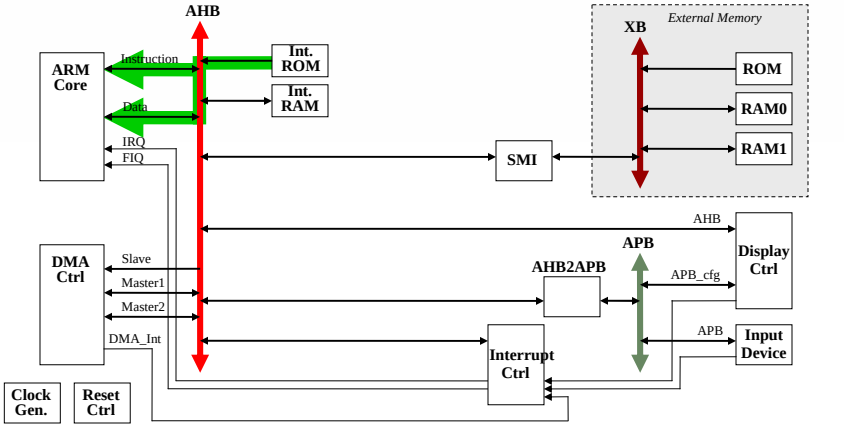
The case study hardware platform consists of the following components



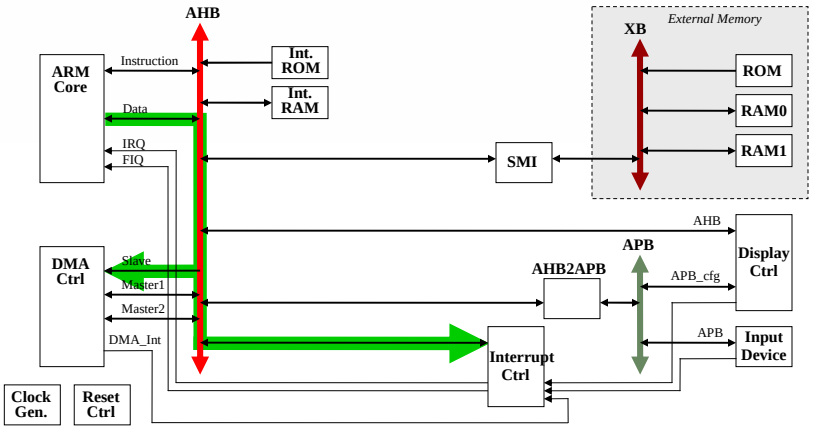
The Application Software includes algorithms to perform Huffman decoding and an inverse discrete cosine transformation (IDCT)



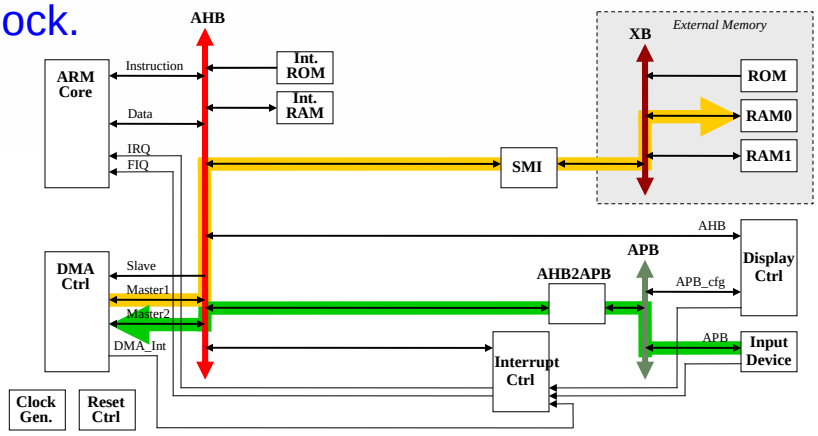
The process begins with the ARM core booting up from the internal ROM with caches and buffers disabled



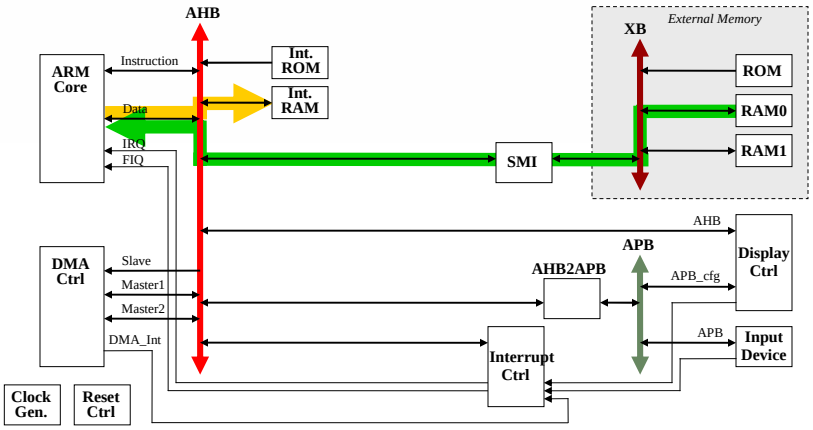
After booting, the ARM core initializes the DMA and interrupt controllers



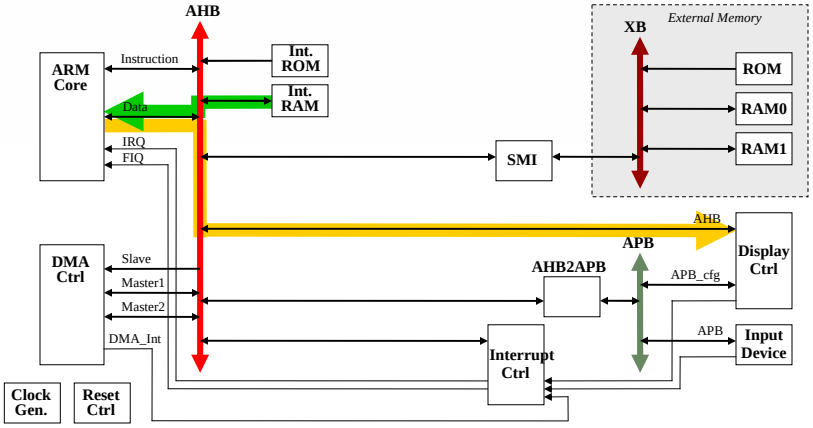
Next the DMA controller copies data blocks of the JPEG picture from the input device to external memory, issuing an interrupt to the interrupt controller after transferring each block.



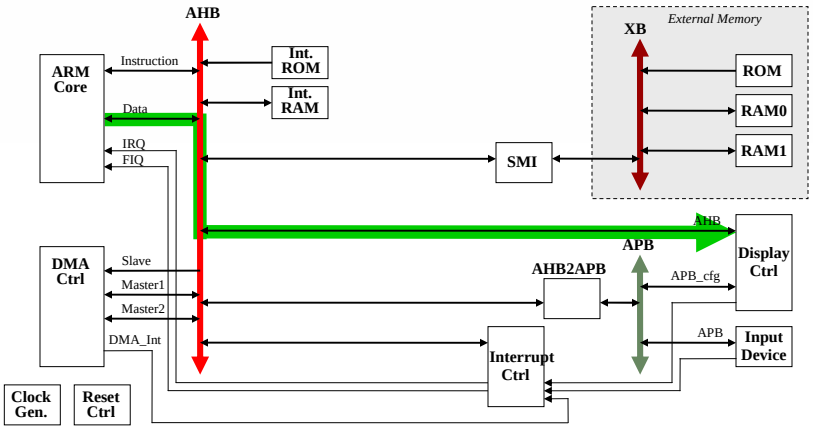
In parallel, the ARM core reads data from external memory, performs Huffman decoding and stores the result in internal memory



The software then performs Inverse Discrete Cosine Transformation (IDCT) on the Huffman decoded data in internal memory, putting the resulting data in the memory of the display controller

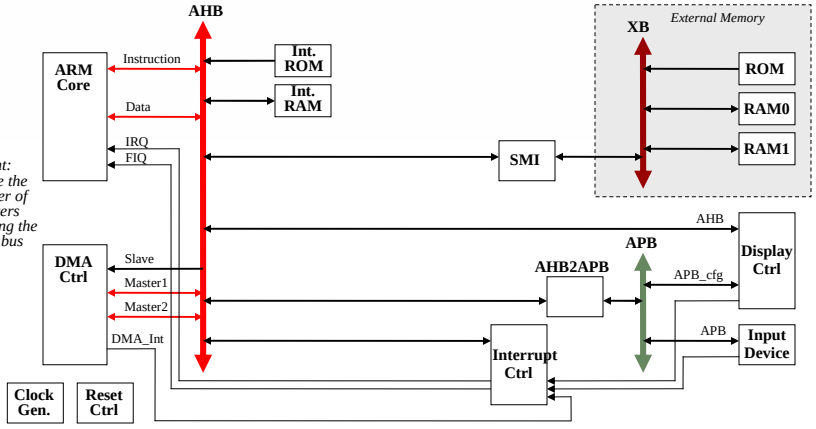


While IDCT data is being stored in the Display memory, the ARM core programs the Display controller. When a complete picture is received, the controller converts it to TIFF and writes it to a file.



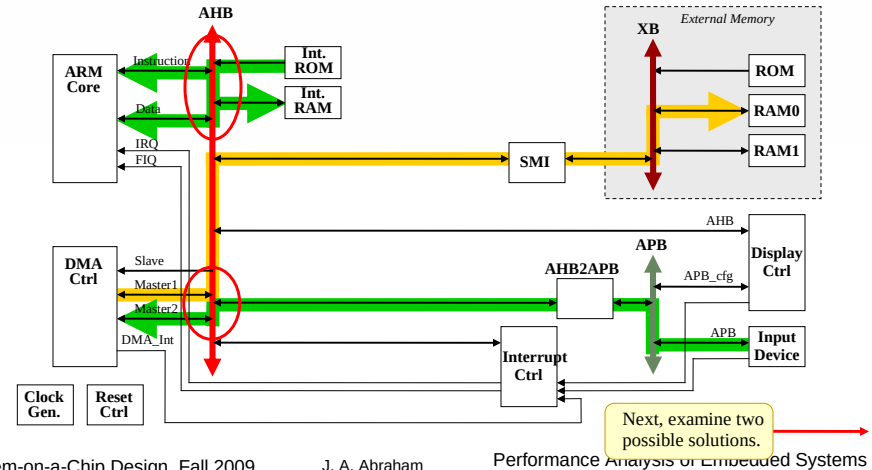
Where do you predict there will be bus contention problems?

Hint: Notice the number of masters accessing the AHB bus

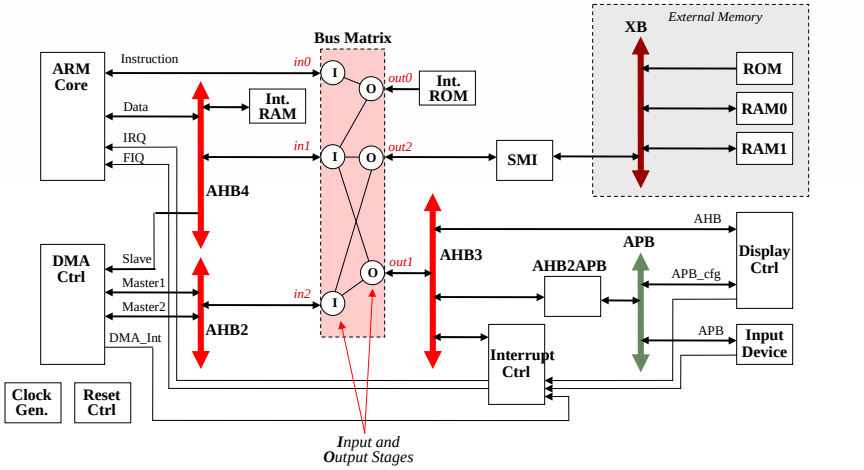


Configuration 1: Contention and utilization problems are primarily due to

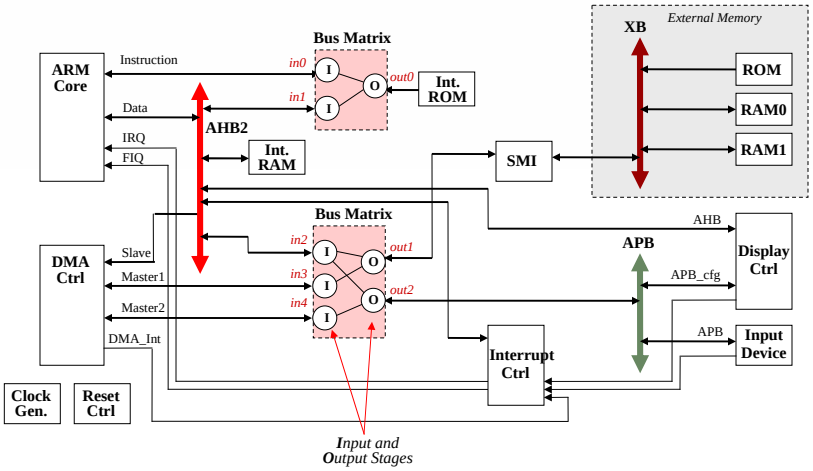
- ARM core to ROM and RAM activity
- Dual DMA activity



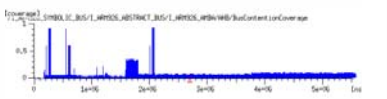
Configuration 2 consists of multiple AHB busses and a multi-layer architecture of Input and Output Stages

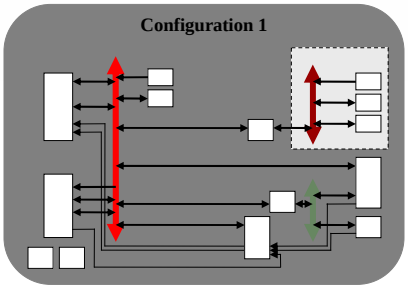


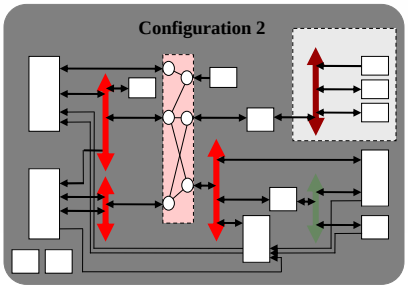
Configuration 3 consists of a single AHB bus and two separate multi-layer architectures of Input and Output Stages

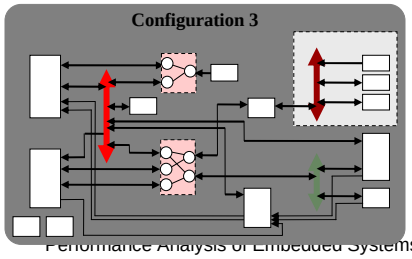


Which configuration will minimize bus contention? What would you predict?









The results for the three configurations are: