CSE/EE 322: Computer Architecture II
Spring 2009

The JAM machine Series

Peter M. Kogge

The Original JAM-8

- Roots in JVM: Java Virtual Machine
  - Target of most Java compilers, & designed for portability
  - Usually executed by SPIM-like JVM interpreter program – but not always
- "Stack-based" ISA
  - Radically different from the MIPS you studied
    - No data register files, just four "pointer" registers
    - Variable length instructions
  - Forces you to really understand "multi-cycle" instruction execution
    - AND memory-intensive computing
  - Will help your understanding of the ubiquitous Java interpreter
- The JAM-8: used since 2000 in Comp Arch II as design target
  - Selected subset of JVM instructions (multiples of 8 bits)
  - Some instructions modified for ease of design
  - All data reduced to 8 bits, with 8 bit addresses (256 byte memory)
  - Rich suite of potential speedup techniques
- Reasons for Change to JAM3D
  - Run programs that consume more than 256 bytes of instructions and data
  - Simplify the design a bit

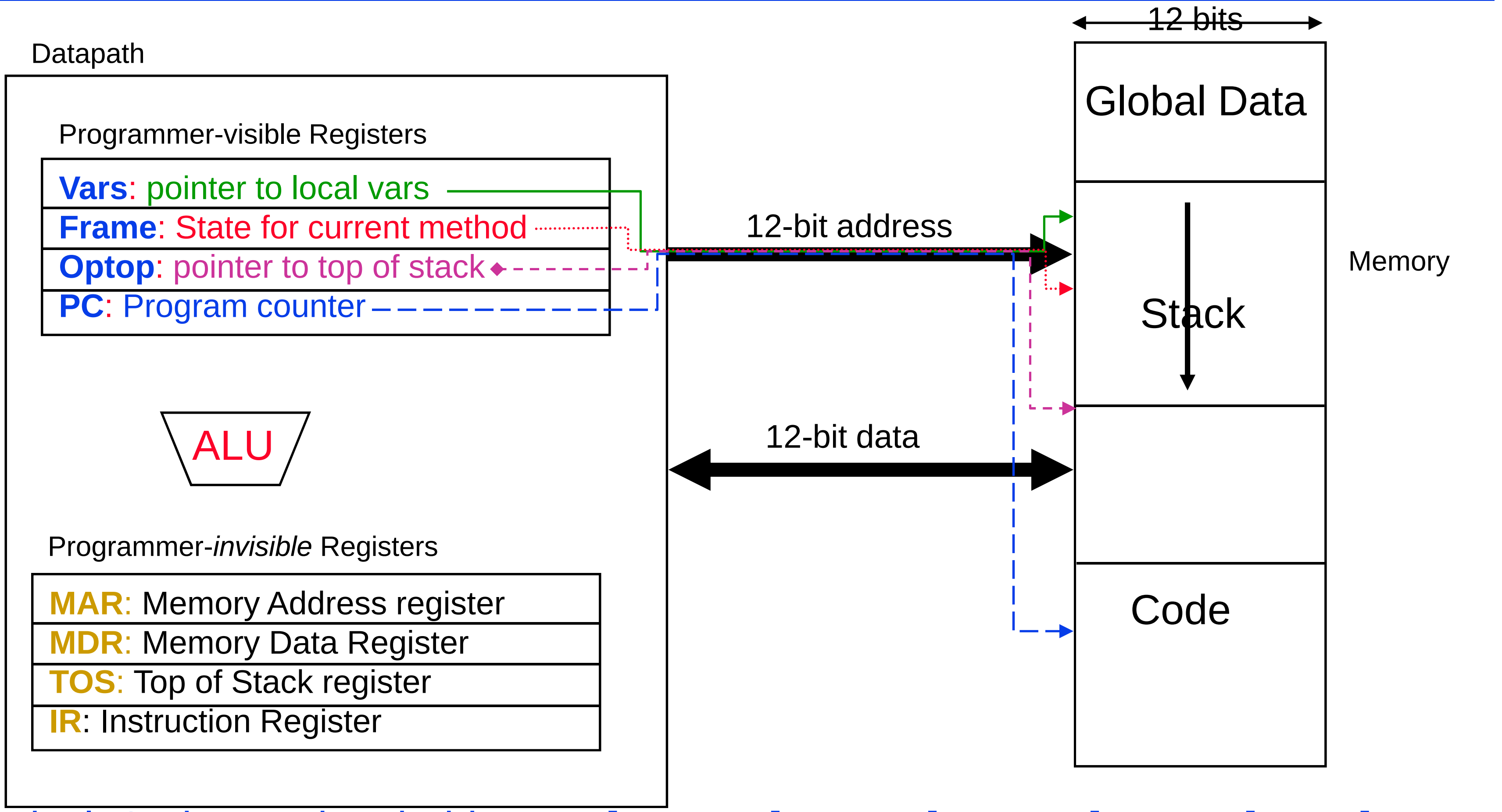
The JAM3D

- Same subset of instructions but re-encoded in 12 bit words
  - I did add a few extra for interpreter only (multiply & divide) that will make for more interesting benchmarks and performance analysis
- All data and memory addresses = 12 bits (4K words of memory)
  - Allows your interpreter to run “looong” programs to gather “real” data
- Only 2 different instruction lengths instead of three
- Enough “space” to allow discussion of ISA extensions
  - Multi-threading
  - Graphics (12-bit word contains three 4-bit R, G, B intensities)
  - …

Why Not a JAM-“16” Instead

- Plus
  - Would have given bigger memory capacity
  - More closely tracked real JVM
  - Matches real memory widths
- Minus
  - Data and instructions now different basic units
  - Requires more complex memory interface
    - Alignment issues
    - Little vs Big Endian considerations
  - And complexity adds little to your ability to design

A Conceptual High Level Picture of JAM3D

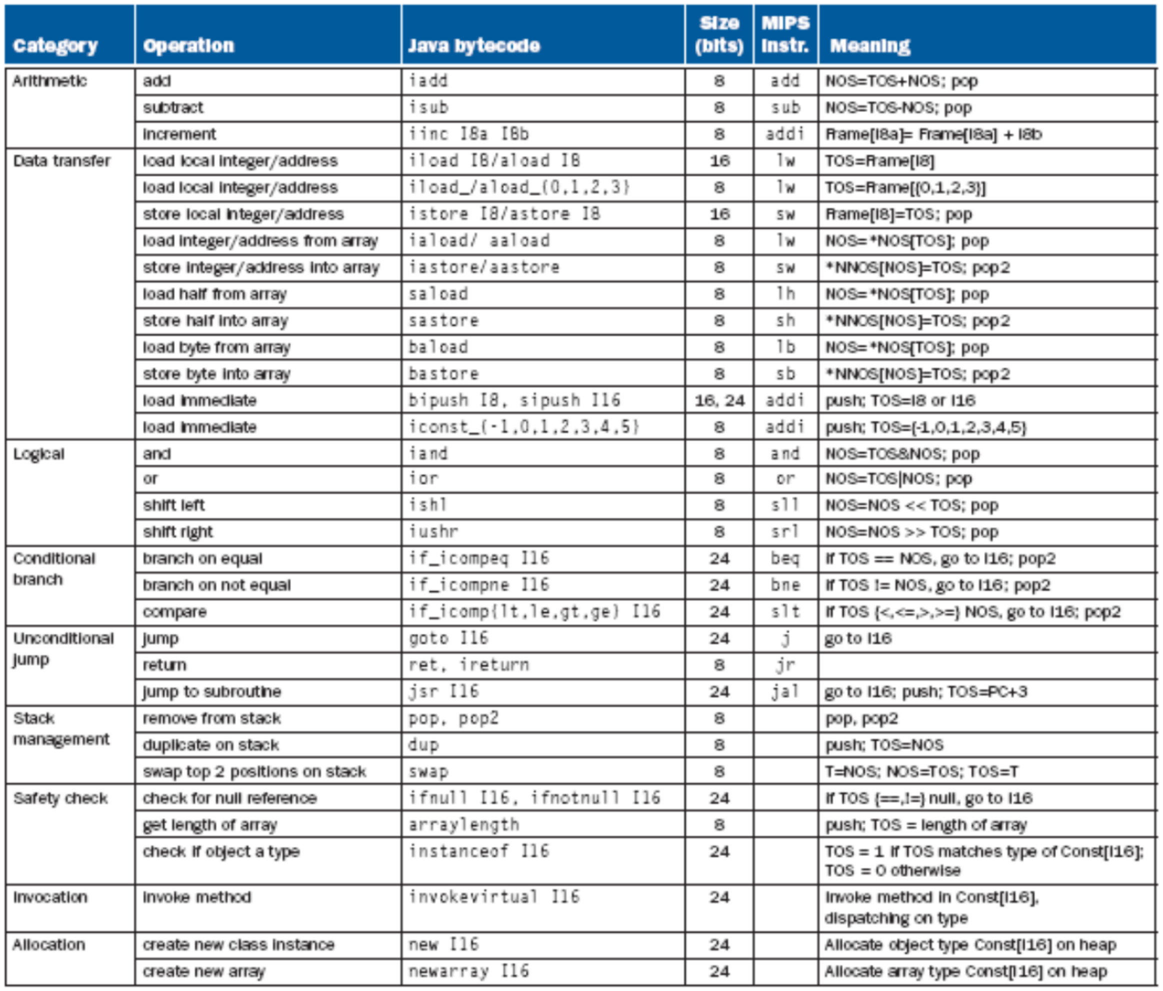


12 bits
Datapath
Programmer-visible Registers
Vars: pointer to local vars
Frame: State for current method
Optop: pointer to top of stack
PC: Program counter
ALU
Programmer-*invisible* Registers
MAR: Memory Address register
MDR: Memory Data Register
TOS: Top of Stack register
IR: Instruction Register
12-bit address
12-bit data
Global Data
Stack
Code
Memory

Typical ISA instruction: iadd: Mem[optop+1]<=Mem[optop+1] + Mem[optop]; optop++

```
Possible implementation: iadd:  MDR = Mem[optop];
                                TOS = TOS + MDR;
                                optop = optop + 2;
                                PC = PC + 1; IR = Mem[PC];
```

A Subset of the JVM from P&H 2.14 (on CD)

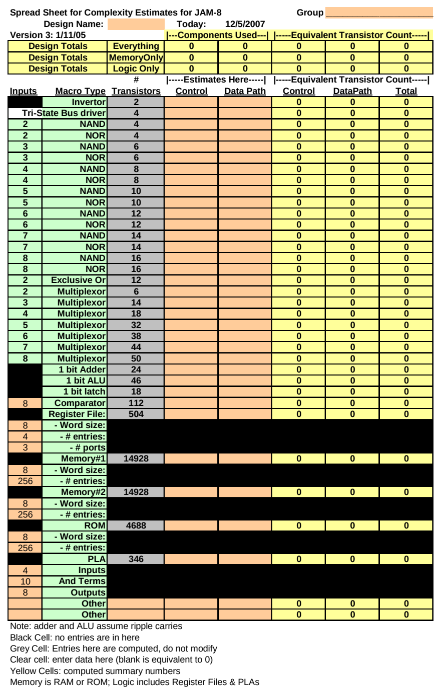


Category	Operation	Java bytecode	Size (bits)	MIPS instr.	Meaning
Arithmetic	add	`iadd`	8	`add`	NOS=TOS+NOS; pop
	subtract	`isub`	8	`sub`	NOS=TOS-NOS; pop
	increment	`iinc I8a I8b`	8	`addi`	Frame[I8a]= Frame[I8a] + I8b
Data transfer	load local integer/address	`iload I8/aload I8`	16	`lw`	TOS=Frame[I8]
	load local integer/address	`iload_/aload_{0,1,2,3}`	8	`lw`	TOS=Frame[{0,1,2,3}]
	store local integer/address	`istore I8/astore I8`	16	`sw`	Frame[I8]=TOS; pop
	load integer/address from array	`iaload/ aaload`	8	`lw`	NOS=*NOS[TOS]; pop
	store integer/address into array	`iastore/aastore`	8	`sw`	*NNOS[NOS]=TOS; pop2
	load half from array	`saload`	8	`lh`	NOS=*NOS[TOS]; pop
	store half into array	`sastore`	8	`sh`	*NNOS[NOS]=TOS; pop2
	load byte from array	`baload`	8	`lb`	NOS=*NOS[TOS]; pop
	store byte into array	`bastore`	8	`sb`	*NNOS[NOS]=TOS; pop2
	load immediate	`bipush I8, sipush I16`	16, 24	`addi`	push; TOS=I8 or I16
	load immediate	`iconst_{-1,0,1,2,3,4,5}`	8	`addi`	push; TOS={-1,0,1,2,3,4,5}
Logical	and	`iand`	8	`and`	NOS=TOS&NOS; pop
	or	`ior`	8	`or`	NOS=TOS|NOS; pop
	shift left	`ishl`	8	`sll`	NOS=NOS << TOS; pop
	shift right	`iushr`	8	`srl`	NOS=NOS >> TOS; pop
Conditional branch	branch on equal	`if_icompeq I16`	24	`beq`	If TOS == NOS, go to I16; pop2
	branch on not equal	`if_icompne I16`	24	`bne`	If TOS != NOS, go to I16; pop2
	compare	`if_icomp{lt,le,gt,ge} I16`	24	`slt`	If TOS {<,<=,>,>=} NOS, go to I16; pop2
Unconditional jump	jump	`goto I16`	24	`j`	go to I16
	return	`ret, ireturn`	8	`jr`	
	jump to subroutine	`jsr I16`	24	`jal`	go to I16; push; TOS=PC+3
Stack management	remove from stack	`pop, pop2`	8		pop, pop2
	duplicate on stack	`dup`	8		push; TOS=NOS
	swap top 2 positions on stack	`swap`	8		T=NOS; NOS=TOS; TOS=T
Safety check	check for null reference	`ifnull I16, ifnotnull I16`	24		If TOS {==,!=} null, go to I16
	get length of array	`arraylength`	8		push; TOS = length of array
	check if object a type	`instanceof I16`	24		TOS = 1 if TOS matches type of Const[I16]; TOS = 0 otherwise
Invocation	invoke method	`invokevirtual I16`	24		Invoke method in Const[I16], dispatching on type
Allocation	create new class instance	`new I16`	24		Allocate object type Const[I16] on heap
	create new array	`newarray I16`	24		Allocate array type Const[I16] on heap

FIGURE 2.14.1 Java bytecode architecture versus MIPS. Although many bytecodes are simple, those in the last half-dozen rows above are complex and specific to Java. Bytecodes are 1 to 5 bytes in length, hence their name. The Java mnemonics use the prefix `i` for 32-bit integer, `a` for reference (address), `s` for 16-bit integers (short), and `b` for 8-bit bytes. We use `I8` for an 8-bit constant and `I16` for a 16-bit constant. MIPS uses registers for operands, but the JVM uses a stack. The compiler knows the maximum size of the operand stack for each method and simply allocates space for it in the current frame. Here is the notation in the Meaning column: TOS: Top Of Stack; NOS: next position below TOS; NNOS: next position below NOS; pop: remove TOS; pop2: remove TOS and NOS; and push: add a position to the stack. *NOS and *NNOS mean access the memory location pointed to by the address in the stack at those positions. Const[] refers to the run time constant pool of a class created by the JVM, and Frame[] refers to the variables of the local method frame. The only missing MIPS instructions from Figure 2.27 are `nor`, `andi`, `ori`, `slti`, and `lui`. The missing bytecodes are a few arithmetic and logical operators, some tricky stack management, compares to 0 and branch, support for branch tables, type conversions, more variations of the complex, Java-specific instructions plus operations on floating-point data, 64-bit integers (longs), and 16-bit characters.

Metrics for JAM3D Designs

- Performance
  - Individual CPI: cycles per each instruction – esp. more “challenging” ones
  - Total cycle count for test program(s)
  - Total instructions executed for test program(s)
  - Average CPI for test programs
  - Perhaps: achievable clock frequency when synthesized for XiLinx
- Cost
  - Number of transistors
  - Computed from spreadsheet of basic logic blocks
- Energy & Power (new)
  - Energy per instruction EPI: cycles * “activity”
  - Energy per program EPP: sum of EPI
  - Power per program: EPP/(cycles*clock)
- ***Cost-Performance*** = #Transistors * average CPI
- ***Power-Performance*** = power per program * time per program



Spread Sheet for Complexity Estimates for JAM-8 **Group** ____________________

Design Name: **Today:** **12/5/2007**

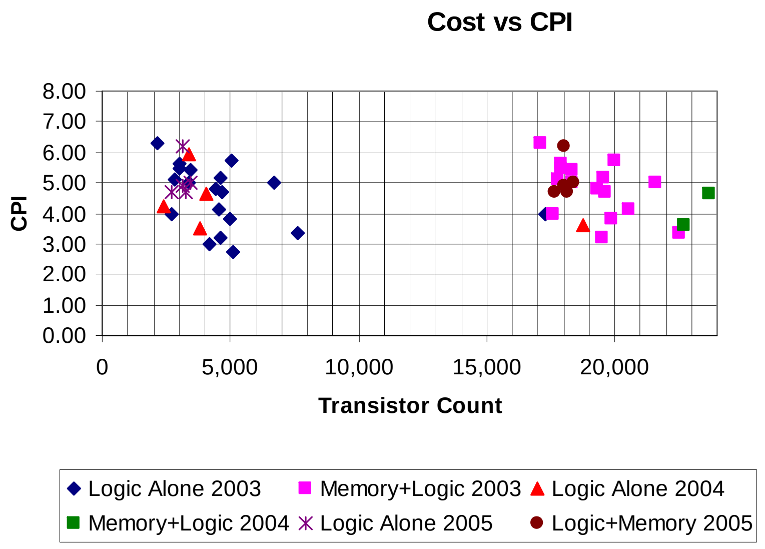
Version 3: 1/11/05

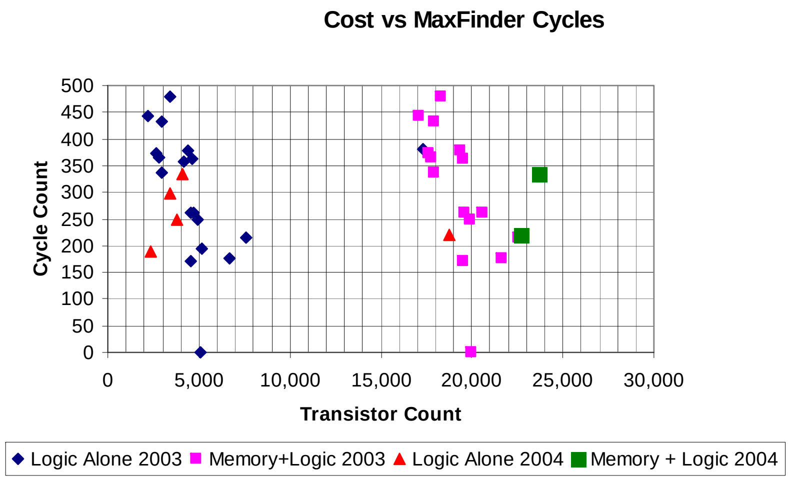
		|---Components Used---|		|-----Equivalent Transistor Count-----|		
Design Totals	Everything	0	0	0	0	0
Design Totals	MemoryOnly	0	0	0	0	0
Design Totals	Logic Only	0	0	0	0	0

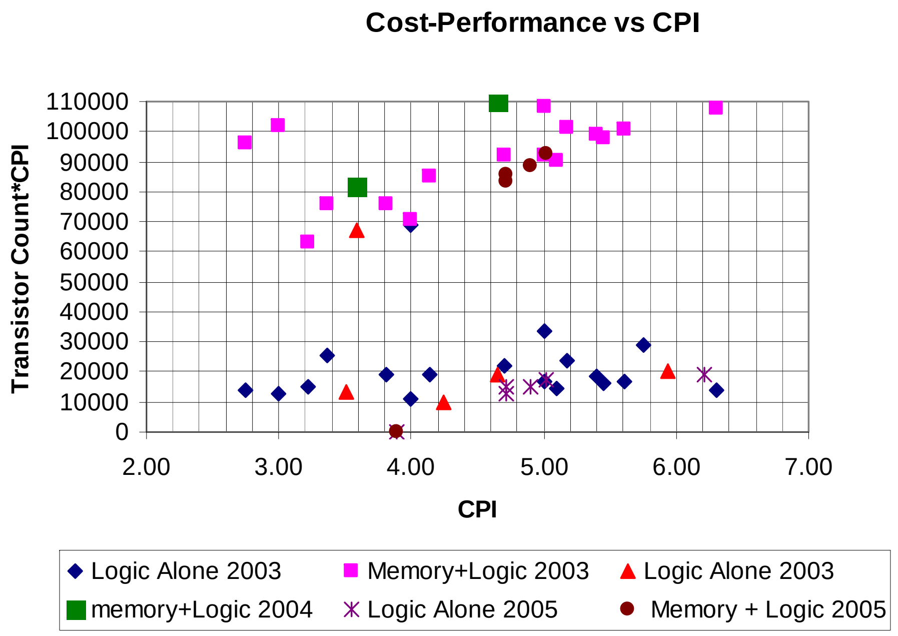
		#	|-----Estimates Here-----|		|-----Equivalent Transistor Count-----|		
Inputs	Macro Type	Transistors	Control	Data Path	Control	DataPath	Total
	Invertor	2			0	0	0
	Tri-State Bus driver	4			0	0	0
2	NAND	4			0	0	0
2	NOR	4			0	0	0
3	NAND	6			0	0	0
3	NOR	6			0	0	0
4	NAND	8			0	0	0
4	NOR	8			0	0	0
5	NAND	10			0	0	0
5	NOR	10			0	0	0
6	NAND	12			0	0	0
6	NOR	12			0	0	0
7	NAND	14			0	0	0
7	NOR	14			0	0	0
8	NAND	16			0	0	0
8	NOR	16			0	0	0
2	Exclusive Or	12			0	0	0
2	Multiplexor	6			0	0	0
3	Multiplexor	14			0	0	0
4	Multiplexor	18			0	0	0
5	Multiplexor	32			0	0	0
6	Multiplexor	38			0	0	0
7	Multiplexor	44			0	0	0
8	Multiplexor	50			0	0	0
	1 bit Adder	24			0	0	0
	1 bit ALU	46			0	0	0
	1 bit latch	18			0	0	0
8	Comparator	112			0	0	0
	Register File:	504			0	0	0
8	- Word size:						
4	- # entries:						
3	- # ports						
	Memory#1	14928			0	0	0
8	- Word size:						
256	- # entries:						
	Memory#2	14928			0	0	0
8	- Word size:						
256	- # entries:						
	ROM	4688			0	0	0
8	- Word size:						
256	- # entries:						
	PLA	346			0	0	0
4	Inputs						
10	And Terms						
8	Outputs						
	Other				0	0	0
	Other				0	0	0

Note: adder and ALU assume ripple carries
Black Cell: no entries are in here
Grey Cell: Entries here are computed, do not modify
Clear cell: enter data here (blank is equivalent to 0)
Yellow Cells: computed summary numbers
Memory is RAM or ROM; Logic includes Register Files & PLAs

JAM-8 Designs: Prior Years Results







Change to Labs

JAM-8 Activities

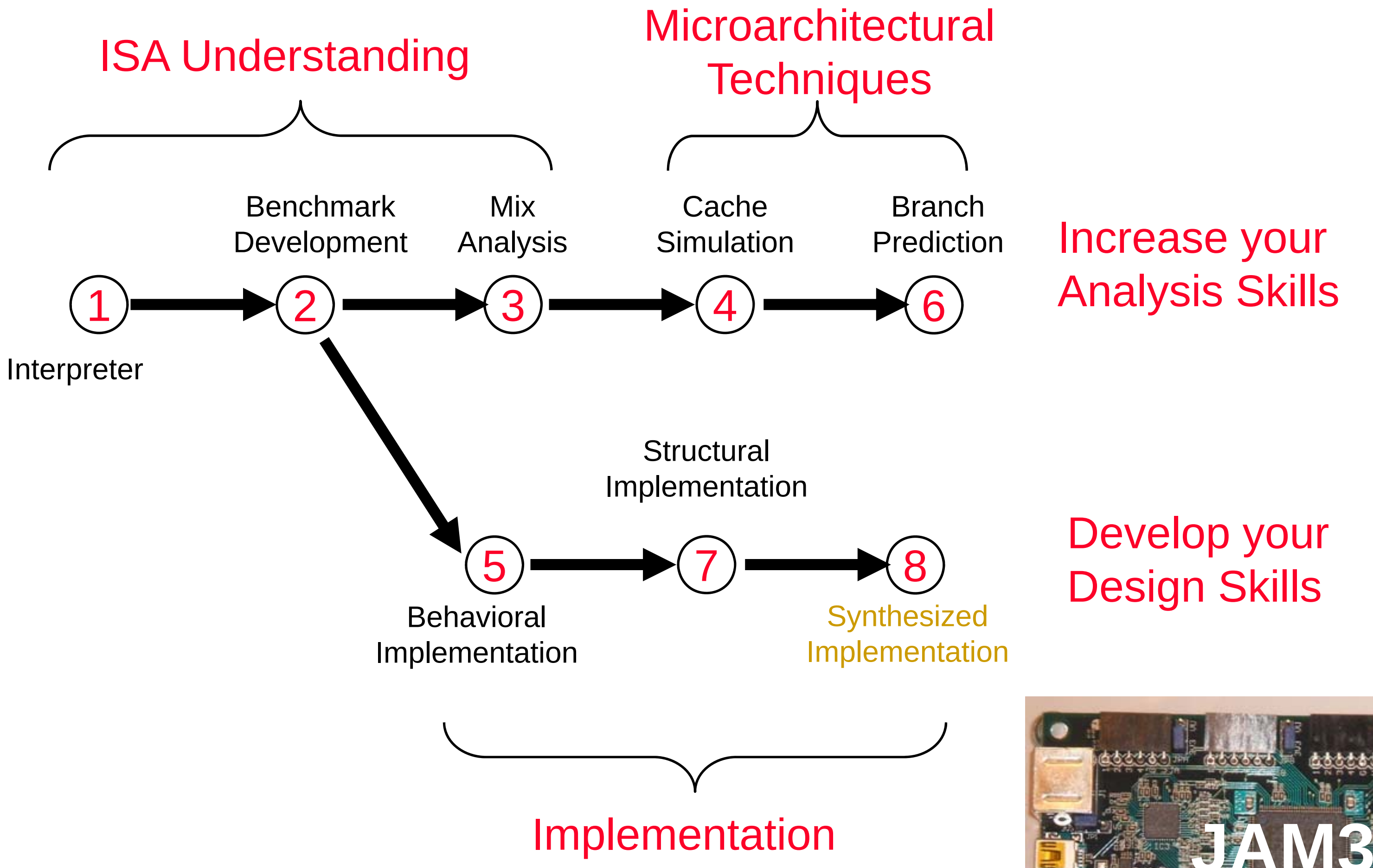
1. JAM-8 interpreter
2. *Cache using Shade*
3. *Pipelining using Shade*: GONE
4. *Branch Prediction using Shade*
5. JAM-8 Behavioral interpreter
6. JAM-8 Structural interpreter

JAM-3D Activities

1. JAM3D interpreter
2. JAM3D Benchmark Development
3. JAM3D Mix Analysis
4. JAM3D Cache Simulation
5. JAM3D Behavioral implementation
6. JAM3D Branch Prediction
7. JAM3D Structural implementation
8. JAM3D on XiLinx: NEW
    1. Includes lab on memory synthesis

Blue Labs build on your JAM3D interpreter
Gold Labs: build towards Xilinx implementation

Lab Component Relationships


ISA Understanding
Microarchitectural Techniques
Benchmark Development
Mix Analysis
Cache Simulation
Branch Prediction
1
2
3
4
6
Interpreter
Increase your Analysis Skills
Structural Implementation
5
7
8
Develop your Design Skills
Behavioral Implementation
Synthesized Implementation
Implementation

Lab Organization

- Meet in Lab every 2 weeks
- Traditional:
  - Go over next lab component
  - Meet as group
  - Get help from TA
  - Use lab facilities (esp. at end)
- New:
  - Collate and discuss results from last component - *in group*
  - Discuss options for next component – *in group*
- Deliverables
  - Comparative data due to TA by Tuesday noon of lab week
  - Lab reports due in class on Thursday of lab week

Non-Design Labs: Benchmark Development

- One or more common benchmarks will be made available
- Each group will select a different short “benchmark”
  - With two data sets: a short “debug” and a long “performance”
  - Convert to JAM3D code
  - Extract statistics from their interpreter
  - Post results to class web site in advance of lab discussion
- Work done in 2 halves
  - 1st Half:
    - benchmark selection and coding (in favorite prog. lang.)
    - Translation to JAM3D assembly, with gathering of static statistics
  - 2nd Half: get running on interpreter (and gather dynamic statistics)
- In Lab discussion
  - Statistics will be combined and compared
  - “Meaning” in terms of impacts on performance will be discussed

Non-Design Labs: Cache interpreter

- Each group will add a simple “cache interpreter” to their ISA interpreter
  - Goal is to measure hit rate as a function of cache parameters
- Common benchmark & individual group program will be run on interpreter with range of cache parameters
  - Again results will be posted to web site
- In Lab Discussion:
  - Again statistics will be correlated and combined
  - Most appropriate cache parameters will be discussed

Non-Design Labs: Branch Predictor

- Each group will add a simple “branch predictor” to their ISA interpreter
  - Goal is to measure prediction rate as a function of design parameters
- Common benchmark & individual group program will be run on interpreter with range of parameters
  - Again results will be posted to web site
- In Lab Discussion:
  - Again statistics will be correlated and combined
  - Most appropriate predictor parameters will be discussed