AMC: An Asynchronous Memory Compiler

Samira Ataei and Rajit Manohar
Computer Systems Lab, Yale University, New Haven, CT 06520
{samira.ataei, rajit.manohar}@yale.edu

***Abstract*—The lack of electronic design automation (EDA) tools for asynchronous circuits makes it challenging to design asynchronous systems that have high complexity. This restriction also makes it almost impossible to prototype and compare new asynchronous designs with their clocked counterparts. The availability of high quality EDA tools would significantly bolster research in asynchronous design, and potentially lead to their adoption in certain application domains.**

This paper presents AMC: an open-source asynchronous pipelined memory compiler. AMC generates SRAM modules with a bundled-data datapath and quasi-delay-insensitive control. AMC is a flexible, user-modifiable and technology-independent memory compiler that generates fabricable SRAM blocks in a broad range of sizes, configurations and process nodes. AMC also produces memory designs that are competitive with both asynchronous and synchronous memories in the literature. AMC aims to reduce design turn-around time and amplify the research in the asynchronous design community.

I. Introduction

Memory latency is a critical performance bottleneck in high-performance digital systems. Data movement between memory and logic (sometimes referred to as the von Neumann bottleneck) largely dominates overall system performance and power consumption in both general-purpose computing and special-purpose architectures for big data computing. Although wide busses can improve memory bandwidth at a cost in area, memory latency remains a performance barrier.

Asynchronous SRAM has the potential to improve SRAM latency and reduce its sensitivity to fabrication variations compared to clocked SRAMs. The standard six transistor SRAM bit-cell produces a differential output, making it compatible with a dual-rail data encoding that encodes both the presence and absence of data. Asynchronous SRAMs can alleviate the timing constraints presented in clocked SRAM, provides higher throughput through pipelining and overlapping memory accesses to different banks, because they naturally handle the variations in memory read access time. Pipelining overhead in the asynchronous case can be designed to minimize impact on read latency, permitting asynchronous SRAMs that have both excellent latency as well as high throughput.

In modern scaled technologies, memory design is consistently one of the most challenging parts of circuit design due to increasing device variability, higher leakage-power consumption, and physical design rule complexity. As a result, the complexity of memory design has dramatically increased. Since most ASICs require some on-chip memory, including memory design as part of individual ASIC development would significantly increase cost. The general practice in industry is to use a third-party memory compiler to minimize development costs, enabling design teams to focus on the core ASIC.

Access to a good memory compiler is a major limitation in academia. Process design kits (PDK) do not include memory blocks, and commercial compilers are not always accessible. This problem is exacerbated for asynchronous chips in both academia and industry, since asynchronous logic is not directly supported by commercial tools or memory compilers.

In recent years, there have been efforts to provide an open-source platform to automate memory layout generation. However, the generated memories by most of such platforms are not silicon-verified and are also not competitive with custom designed high-performance memories. Synopsys' generic memory compiler [1] only supports Synopsys' 32/28nm and 90nm abstract technologies, which do not correspond to a foundry technology.[1] OpenRAM [2], developed through a collaboration between OSU and UCSC, is a synchronous open-source memory compiler. While OpenRAM provides a flexible and portable platform to implement clocked memories, it doesn't support thin-cell layout (needed for the design rules at 65nm and below) or modern design rules. Also, the performance of memories generated by OpenRAM is low as the sequence of operations needed for memory operation (capturing the address and data bits, address decoding, bitline precharging and data driving/sensing) are bounded by clock edges which does not allow memory to perform at its highest frequency. With only four banks, the biggest 32-bit memory that can be generated by OpenRAM compiler is 32KB.

This paper presents AMC: an open-source asynchronous pipelined memory compiler. AMC applies techniques from pipelined asynchronous circuit design to improve the average-case performance of memory access and supports non-uniform memory access time. AMC generates multi-bank SRAM modules, using a combination of quasi delay-insensitive (QDI) control and bundled-data datapath. The largest 32-bit memory that can be generated by AMC is 1 MB. AMC supports an atomic read-modify-write operation in addition to the read and write memory operations that has a longer cycle time compared to read/write operations, exploiting the ability of asynchronous circuits to handle data-dependent delay. AMC leverages the micro-architectural techniques introduced by the cache of an asynchronous MIPS processor [3], and inherits the data structures and base functions from the OpenRAM compiler [2]. Like OpenRAM, the back end GDSII frame-

[1]This enables them to provide the technology information without involving complex foundry non-disclosure agreements.

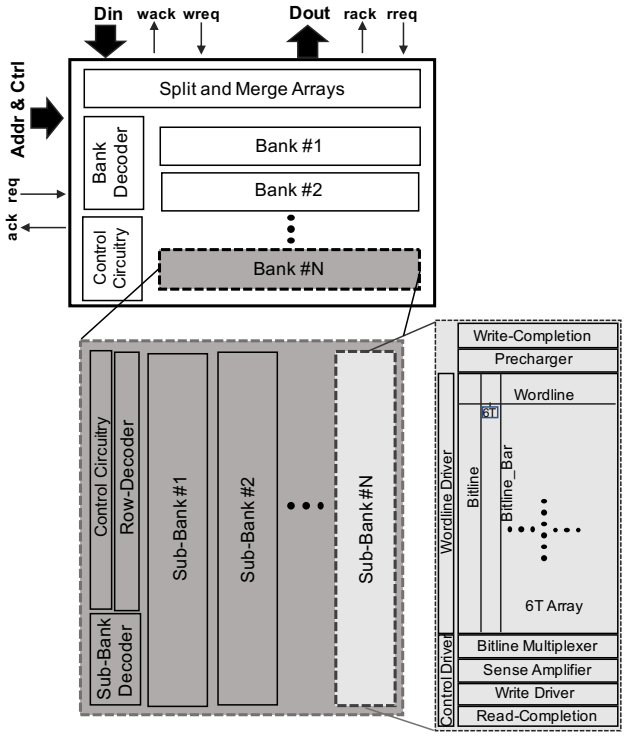

Fig. 1. An AMC multi-bank architecture with input and output channels. Each bank consists of sub-banks, decoders and control circuitry. Each sub-bank is compose of 2D bitcell array, read and write control circuits, completion detection, drivers and decoders.

work uses GDSMill [4] for scripting layouts in the Python programming language. AMC has been written so that its core is technology-independent, with parameters and layout fragments capturing technology-specific components. AMC generates circuits, GDSII layout data, SPICE netlists, and Verilog models for SRAM in variable sizes, configurations and process nodes. At the moment of writing, AMC can generate memories for both scalable CMOS (SCMOS submicron λ rules) and 65nm, both technologies that can be used to fabricate chips. A test chip is under development to validate the compiler. Post-layout SPICE simulations show that the memories generated by AMC are competitive with published designs.

AMC is distributed with a reference design for SCMOS technology, as this is the last technology node that has open-source design rules. The distribution is freely available at:

`https://github.com/asyncvlsi/AMC`

The rest of this paper is organized as follows. Section II explains the self-timed interface of asynchronous SRAM implemented by our compiler, and the basic structure of the memories generated. In Section III, AMC implementation and compiler features are explained. Simulation results, sample layouts and detailed comparisons in a $0.5\mu m$ technology (with OpenRAM [2]) and a 65nm technology (with Fulcrum Microsystems memories[5]) are shown in Section IV. Section V outlines the ongoing and future work for this compiler and concludes the paper.

II. Memory Design With QDI + Bundled-data Logic

AMC uses conventional asynchronous design techniques to generate QDI + bundled-data logic memories. The QDI methodology assumes that gates have arbitrary delay and only makes relatively weak timing assumption on the presence of isochronic forks [6], [7]. This results in control circuits that are tolerant to PVT variations.

The external interface of the multi-bank SRAM generated by AMC is shown in Figure 1. In this figure, *Addr* is the input address bus, *Ctrl* is the input control bus, *Din* is input data bus and *Dout* defines the output data bus. The bundled-data encoding in the AMC interface uses multiple sets of independent request/acknowledge pairs: one set for *Din* (called `wreq`, `wack` in Figure 1); one set for *Dout* (called `rreq`, `rack`); and a third set for both the address and control (called `req`, `ack`).

AMC uses the Martin synthesis method [6] to design all the QDI control logic. This synthesis method allows us to describe circuits algorithmically using the CHP (Communicating Hardware Processes) language, briefly described in the Appendix. The following CHP program describes the set of operations executed by AMC SRAMs:

$$
\begin{aligned}
&MEM \equiv \\
&\quad *[\ Addr?a,\ \ Ctrl?c; \\
&\quad\ [c = \text{READ} \longrightarrow Dout!mem[a] \\
&\quad\ [] c = \text{WRITE} \longrightarrow Din?mem[a] \\
&\quad\ [] c = \text{RMW} \longrightarrow Dout!mem[a]; Din?mem[a] \\
&\quad\ ]]
\end{aligned}
$$

The address received on channel *Addr* defines the row and column access location in the 2D bitcell array. The *Ctrl* defines the type of operation which can be read, write or read-modify-write (RMW). *Addr*, *Ctrl* and *Din* are the input channels and *Dout* is the output channel to SRAM bank.

AMC uses four-phase handshaking (return-to-zero) with the data and address encoded as bundled data. In AMC, a single memory bank is implemented as one process that synchronizes the handshake on the appropriate input and output channels needed for the specified operation. A single asynchronous memory bank can be described as a process that receives the address and control inputs, performs the read or write operation, and possibly transmits data to its environment. A number of such memory banks together makes up a pipelined asynchronous multi-bank memory system.

A. Core Bank and Sub-banking

A bank consists of a collection of *sub-banks* that share row decoders. Figure 2 shows the major components of the memory access circuitry within a sub-bank. The six transistor (6T) SRAM cells are organized into a two-dimensional array, with wordlines and bitlines in the usual way. Each sub-bank has a local sub-bank select signal, that is combined with the bank wordline to generate the local wordline for the sub-bank. Apart from the hierarchical wordline structure, the rest of the

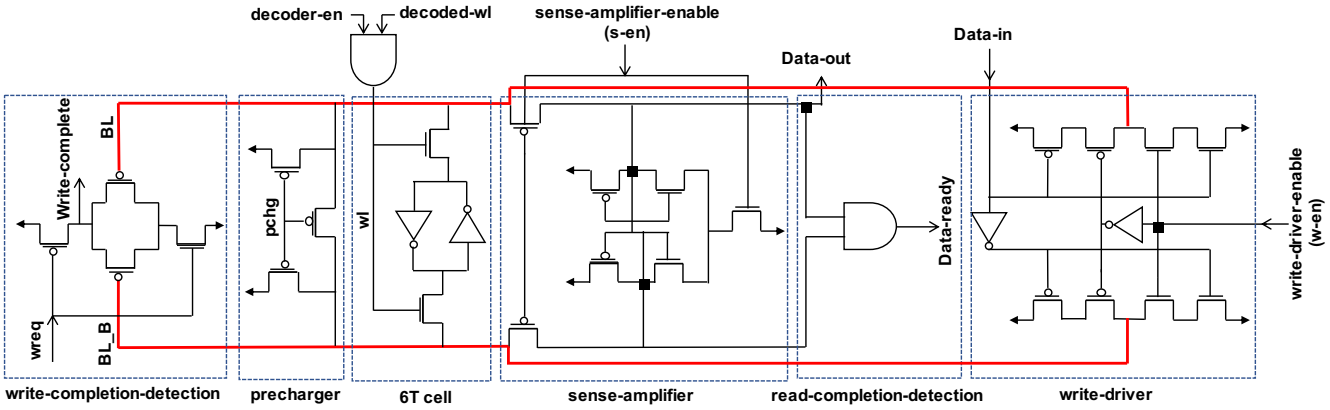

Fig. 2. Bitline datapath circuitry

sub-bank circuitry is similar to standard SRAM design. The two bitlines can be used to determine when valid data has been received, similar to previous designs [3].

B. Banking and Pipelining

The memory sub-bank consists of an array of bitcells. These bit cells are carefully optimized to minimize area, and reliable, high-performance memory operation is only attainable as long as the size of the bitcell array is limited. To increase the total capacity of the memory, multiple such banks are needed. In a standard clocked memory, banking is used to improve the performance of a memory. As long as the latency of the access are sufficiently short, banking can improve performance. However, beyond a certain size, banking impacts both the latency as well as the cycle time of a single cycle clocked memory.

AMC creates a memory with a pipelined asynchronous architecture, similar to multiple previous custom memory designs [3], [8], [9], [5], [10]. The banks are organized into a tree structure, and small memory banks are located at the leaves of the tree. The structure of the tree can be specified by the user, and can use a branching factor of two or four at each level. Figure 3 shows a two-level tree with branching factor of four at both levels. The inner bus controls access to up to four memory banks, and the outer bus controls access to up to four copies of the inner buses. The split and merge circuits used to implement the tree are pipelined, permitting access to a different bank while a previously accessed bank is still performing an operation [3], [5]. AMC creates the appropriate floorplan and physical design based on the banking options.

To be able to pipeline the memory, split bus for the input data, address and request controls and merge bus for the output data and acknowledge controls are added for each memory bank. The selection between banks is accomplished through a bank decoder. The split is a 1 to 4 bus which reads a bank select control input from the bank decoder and input data from write channel *Din*. It then routes the data to one of the output channels selected by the control input. The merge is the dual of this: it is a 4 to 1 bus which reads a bank select control from the bank decoder and reads a data value from one of the input channels selected by the control input, and sends that data to a single output channel *Dout*.

This can be summarized by the following CHP program:

$$
\begin{aligned}
&MEM \equiv \\
&\quad *[\ Addr?(bank, a), Ctrl?c; \\
&\quad BCtrl[bank]!c, BAddr[bank]!a \\
&\quad [c = \text{READ} \longrightarrow BDout[bank]?dout; Dout!dout \\
&\quad []c = \text{WRITE} \longrightarrow Din?din; BDin[bank]!din \\
&\quad []c = \text{RMW} \longrightarrow BDout[bank]?dout; Dout!dout; \\
&\quad Din?din; BDin[bank]!din \\
&\quad]]
\end{aligned}
$$

Figure 3 shows a basic block diagram of 16-bank SRAM with split and merge buses. In the inner bus, split includes one split environment part, contains the split control circuitry, and four split cell-array parts which are replicated for each output channel. Merge includes one merge environment part and four merge cell-array parts. The use of pipelined split and merge buses allows the multi-bank SRAM to operate at a higher throughput compared to a single monolithic bank.

In synchronous SRAMs such as memories generated by OpenRAM, only one bank can be active at a time and there is no pipelining between the operations of the banks. However,

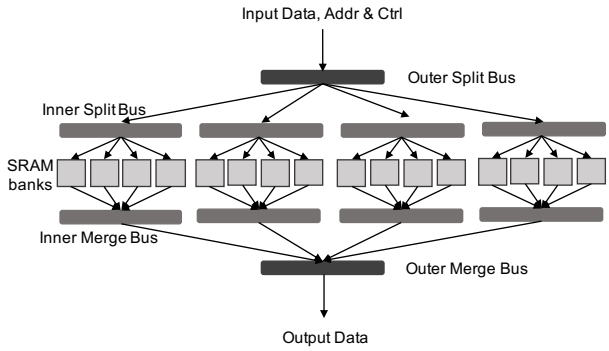

Fig. 3. A two-level split-merge bus. The control input for the split and merge blocks is not shown.

TABLE I
SUPPORTED CONFIGURATION OPTIONS BY AMC

Word Size	>= 1 bit
No. of Rows	>= 16
No. of Words per Row	1, 2 and 4
No. of Sub-Banks	1, 2, 4 and 8
No. of Banks	1, 2, 4, 8 and 16
Banks' Orientation	Horizontal and Vertical
Technology Node	e.g. SCN3M_SUBM

AMC generates pipelined multi-banked memories and permits multiple banks to be active at the same time based on the memory access pattern. This enables construction of SRAMs with arbitrary sizes at constant peak frequency. Access time results reported in Section IV verify this observation.

The asynchronous memory generated by AMC has timing behavior that depends on the memory access pattern. This is in contrast with synchronous SRAM where clock frequency is determined by the propagation delay of critical path in a bank (normally read-access delay). In synchronous SRAMs, the same path during write operation, that normally takes less time to complete, is idle most of the time in a clock cycle and cannot go faster than the expected worst-case clock frequency. However, in asynchronous memory, a latency increase for read operation would not affect the latency of write operation. We exploit this property to include a read-modify-write cycle in which the address decoders are used just once. Section IV shows that this operation is only slightly slower than a read. This results in an improvement for typical operations in a multi-bank asynchronous memory. This non-uniform memory latency does not add any complexity to asynchronous memory controller because its circuits are designed to handle uncertainty in response times.

III. AMC METHODOLOGY

AMC is implemented in an object-oriented approach in the Python programming language. AMC uses the data structures of OpenRAM and inherits some of its utility functions. To read/write and manipulate GDSII files in Python, AMC uses GDSMill package [4]. AMC generates GDSII layout data, standard SPICE netlists, Verilog models, and DRC/LVS verification reports of different SRAM sizes and configurations. AMC can be ported to any technology node by including technology-specific rules and pre-made library cells. At the moment of writing, AMC has been successfully ported to a 65nm process and SCN3M_SUBM $0.5\mu m$ technology. All physical implementations and information in SCN3M_SUBM $0.5\mu m$ technology, which is fabricable through the MOSIS VLSI service [11] is part of the compiler as a reference implementation.

Figure 4 shows the AMC flow. The compiler generates SPICE and Verilog netlists, GDSII layouts, functional and physical verification reports and abstract place and route LEF file. AMC also includes a memory characterizer to generate the timing and power reports for synthesis. The timing and power results are generated with the following steps: (i) generating the stimulus, (ii) calling a SPICE simulator and running the circuit simulations in batch mode, (iii) parsing the outputs, and (iv) producing standard lookup-table format Liberty (.lib) file. Extracted spice netlist generated from the GDSII layout can also be used to make annotated timing and power models.

Table I shows the options that are available for configuration by the user. To make the compiler easy to use, all the options can be specified through a single configuration file or through command line options.

This compiler allows a user to add new technology libraries. To port AMC to a new technology node, custom designed library cells such as the 6T SRAM cell, sense amplifier and write driver must be added to the compiler. For reasons of density these cells have to be manually designed for each technology. It is also possible to use foundry-specific SRAM cells and non 6T cell (e.g. 8T or 10T cells). Parametrized cells such as transistors, inverters and pull-up pull-down switching networks are generated by the compiler. The compiler also dynamically generates different sizes of inverters that are pitched-match with the manually designed cells. In addition, design rule check (DRC) rules and the GDS layer map for the technology node are required to ensure that the generated SRAM layout is DRC clean.

AMC's memory architecture as shown in an abstract view in Figure 1 is based on multi memory banks, control circuity, decoder and split/merge arrays (details in Section II-B). An SRAM bank consists of multiple sub-banks each composed of bitcell array, row-address decoder, column-address decoder, bitline multiplexer, wordline driver, sense amplifier, write driver, precharger, read and write completion circuitry and control logic part. AMC tiles the bitcells together to make the 2D SRAM bitcell array. Bitline and wordline connections are made through abutment. Sense amplifier, write driver, precharger and completion detection circuitry are vertically pitch-matched with the bitcell, and bitlines in all these modules are connected by abutment. Also, the row-address decoder and wordline driver are horizontally pitched-match with the bitcell, and the wordline in all these modules connect by abutment. AMC accepts user-provided parameters (number of rows, data word size, and number of banks and sub-banks),

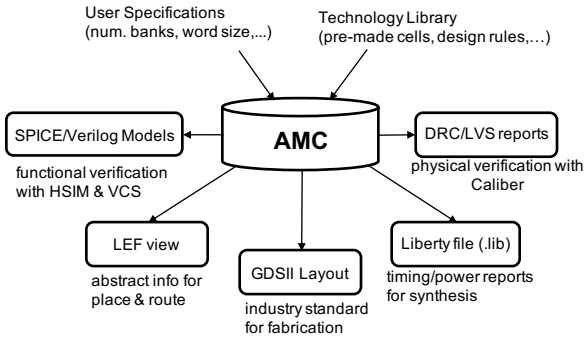

Fig. 4. AMC flow

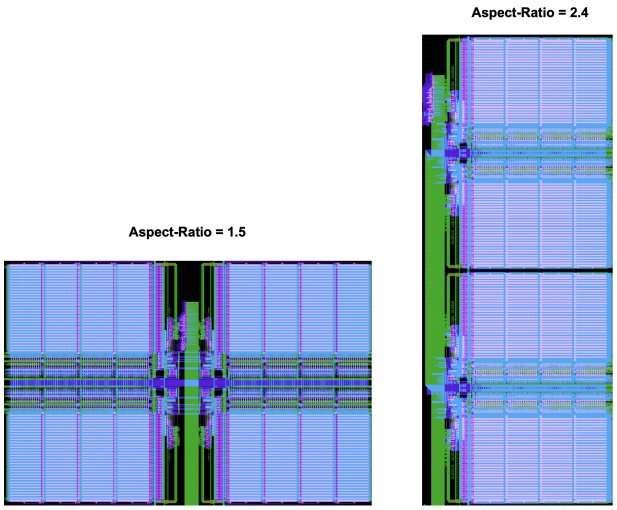

Fig. 5. SRAM banks are place in different orientations resulting in different aspect ratios for same size SRAM module.

physically places and logically connects all the modules to generate a multi-bank SRAM. AMC leverages OpenRAM's testing infrastructure, and has unit tests for each sub-module to guide users when porting the compiler to new technologies. These unit tests allow users to add features and simplify debugging when switching to a new technology.

The following subsections describe features of AMC which were important for the performance and density of its generated SRAMs:

A. Partitioning and Floorplaning

In AMC each bank encapsulates multiple sub-banks of SRAM arrays. Each sub-banks is an array of SRAM cells along with peripheral circuitry, while all sub-banks share one row-decoder. A row-decoder decodes the input address and generates the global wordlines that are gated with a sub-bank select signal to create the local wordlines. This design feature allows us to break the SRAM array into smaller width segments and enables a lower access time by reducing the RC delay of the wordline. It also enables bigger SRAM banks without increasing the access time. Memories with smaller sub-partitions also help to save energy as only one partition is active at any time.

B. Bank Orientation

In AMC, banks can be placed in different orientation to get different aspect ratios for the SRAM. Compared to hard multi-bank layout with fixed aspect ratio generated by traditional compilers, AMC generates soft multi-bank layouts which results in an optimum floorplan. This feature allows designer to add the best matching SRAM layout to the rest of their design. Figure 5 shows four banks placed in different orientations but that have the same top-level interface.

C. Memory Operations

AMC adds a third type of operation compared to conventional SRAMs, namely a read-modify-write cycle. This operation performs simultaneous read and write at a small cost compared to an individual read operation, and with less time and energy compare to two separate operations. The read-modify-write operation allows memory to read a location and write it with a new value simultaneously while activating the decoder just once. Such an operation would increase the clock cycle period in a synchronous memory, but in the asynchronous case it only increases the cycle time when the operation is used.

D. SPICE/Verilog co-simulation

The files generated by AMC include both a SPICE and Verilog netlist. These blocks can be used to model the memory in the rest of the user's design. To get faster verification for large memory blocks, AMC adds a functionality unit test that co-simulates the Verilog and SPICE netlists. This co-simulation keeps part of the design at the digital level leading to faster simulation runs compare to a transistor-level netlist simulation in SPICE. This Verilog-SPICE integration uses a Verilog test-bench to drive an SPICE simulation of the SRAM at the transistor-level. It provides a top-level verification with many random input vector patterns while SRAM block is simulated in SPICE. This co-simulation is extremely useful when an SRAM block is integrated with other digital logic described in Verilog.

IV. Evaluation

Figure 2 shows the bitline datapath circuitry used in the design of SRAM. In AMC access time of each operation is measured as the time interval of consecutive operations. The sequence of transitions for one cycle of read operation is as follows:

1) Read operation starts when both read-control and read-request signals go high by environment. ($r\uparrow, rreq\uparrow$)
2) As soon as the operation starts, precharging stops by the controller ($pchg\uparrow$)
3) Decoder-enable goes high by the controller and triggers the wordline ($decoder-en\uparrow, wl\uparrow$)
4) Sense-amp enable signal, generated by a dummy bit-line column to mimic the delay of bitline, goes high ($s-en\uparrow$)
5) Read-completion signal, generated by ANDing the data_out and data_out_bar of differential sense-amplifier, goes high when full swing data is ready on the output data bus ($data-ready\uparrow$)
6) Read acknowledge signal is generated by controller and sent to environment ($rack\uparrow$)
7) Environment enables the acknowledge signal and lowers the read-control and read-request ($ack\uparrow, r\downarrow, rreq\downarrow$)
8) Controller starts the precharge for the next operation cycle when read-control signal is lowered ($pchg\downarrow$, $rack\downarrow$)
9) Controller returns the acknowledge signal to zero and completes the handshake ($ack\downarrow$)

For write operation (w) same sequence of transitions repeats with write-request (*wreq*), write-acknowledge (*wack*) and *write-complete* signals. In read-modify-write case (*rmw*), after

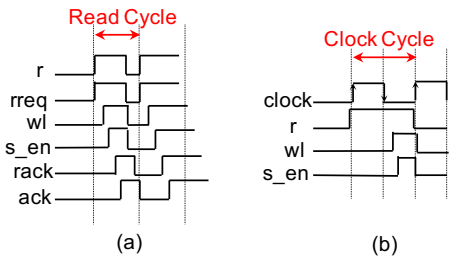

Fig. 6. Read cycle in (a) asynchronous and (b) synchronous SRAM.

read acknowledge enables in step 6 ($rack \uparrow$) write operation starts ($wreq \uparrow$) and handshake completes when write is finished ($wack \downarrow$).

In AMC each of the three operations (*r, w, rmw*) has different access time compared to the synchronous OpenRAM where each operation takes exactly one clock cycle. Figure 6 compares the read cycle time of asynchronous and clocked SRAMs. In synchronous SRAM such as those generated by OpenRAM compiler, wordline and sense-amp enable (*s-en*) signals are gated in negative edge of the clock, hence, reading the cell's stored-value occurs in the second half of the cycle while first half is used for precharging. Normally, reading the stored value through discharging the heavily loaded bitline takes more time than charging the bitline with a precharge circuitry. When AMC is configured to create a single bank without sub-banking, the larger cycle time of synchronous OpenRAM memory compared to AMC-generated memory can be attributed to using a 50% duty-cycle clock, combined with satisfying the setup-time and hold-time constraints on flip-flops in the synchronous memory.

In the following subsections, the SRAM generated by AMC is compared with both synchronous and asynchronous SRAMs to show the effectiveness and quality of memories generated by the AMC compiler. Reported timing and power values for AMC are based on post-layout simulations. AMC generates the GDSII layout and a SPICE netlist including all the parasitic resistance and capacitance extracted from the layout using Calibre extraction tool. The extracted netlist is then simulated with Synopsys' SPICE simulator. All performance and power results for OpenRAM-generated and AMC-generated memories used this methodology. Results for commercially designed memories were obtained from published literature.

A. Comparison with Synchronous SRAMs

Multiple size and configurations of generated memories by OpenRAM compiler (as an accessible synchronous memory compiler) and AMC are simulated in SCN3M_SUBM $0.5\mu m$ technology and compared in Table II for access time, average power consumption, layout area and layout efficiency. Layout efficiency can be defined as the amount of area dedicated to 6T cells versus the total area including all the control circuitry. The efficiency quantifies the control overhead of the design.

Table II shows three cycle time values for AMC SRAMs: read, write, and read-modify-write. In AMC, each operation takes minimum possible time to complete unlike OpenRAM where slow read operation dictates the clock frequency for both operation modes. In OpenRAM only one bank is active at each clock cycle and there is no pipelining and overlapping between banks. Also, since AMC uses a three-level partitioning (multi-bank, bank, and sub-bank) compared to two level partitioning in OpenRAM, bigger SRAMs can be generated. We compare AMC with two options: Option I doesn't use sub-banking, whereas Option II does. Option I SRAMs are smaller in area compared to option II, but are slower because of increased wordline loading (Table II). AMC Option I SRAM bit efficiencies are very close to OpenRAM's bit efficiency, while performance is 1.2X-2.0X better. Option II SRAMs provide even higher performance (1.5X-2.35X) and also lower power consumption (0.56X-0.78X) SRAMs compared to OpenRAM at an additional area cost (1.14X-1.34X).

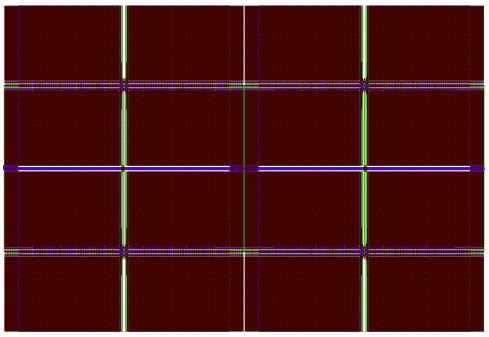

Fig. 7. 1MB SRAM generated by AMC in SCN3ME_SUBM $0.5\mu m$ technology. This 32-bit SRAM contains 16 banks where each bank has 8 sub-bank and each sub-bank is a 512 x 128 bit array. Total area is $1437mm^2$.

With more than one bank, AMC can overlap the operations of the banks. We have the choice of mapping the banks based on the address. Table II reports best-case (switching accesses between banks) and worst-case (consecutive accesses to one bank) cycle time. It was previously shown that low-order banking (where the least significant bits of the address are reserved for selecting the bank) rarely results in consecutive accesses to the same bank [8].

The maximum number of supported banks by OpenRAM compiler is four, banks while AMC can generate up to sixteen banks. Figure 7 shows the layout of a 2MB SRAM generated by AMC in SCN3ME_SUBM $0.5\mu m$ technology.

B. Comparison with Asynchronous SRAM

We compare AMC-generated memories to custom SRAMs designed by Fulcrum Microsystems [5]. This comparison allows us to evaluate the quality of AMC SRAMs in a fabricable 65nm process node. A comparison in this process node with OpenRAM is not possible as OpenRAM doesn't support thin-cell layout (height of decoder and wordline driver cannot be pitch-matched with SRAM cell height). Figure 8 shows both tall-cell (in SCN3ME_SUBM $0.5\mu m$) and a DRC-clean thin-cell layout (representative of a 6T SRAM thin-cell in 65nm) which were manually created to be used by AMC.

TABLE II
COMPARISON FOR DIFFERENT SRAM SIZES AND CONFIGURATIONS IN SCN3M_SUBM 0.5μm

Configuration				OpenRAM Compiler				AMC Compiler							
								Option I (without sub-banking)				Option II (with sub-banking)			
No. of Bank	**Word Size** (bit)	**Word per Row**	**No. of Rows**	**Cycle Time** (ns)	**Avg. Power** (mW)	**Total Area** (mm^2)	**Bit Efficiency** (%)	**Cycle (r, w, rmw)** (ns)	**Avg. Power** (mW)	**Total Area** (mm^2)	**bit Efficiency** (%)	**Cycle (r, w, rmw)** (ns)	**Avg. Power** (mW)	**Total Area** (mm^2)	**bit Efficiency** (%)
1	8	1	32	6.4	13	0.19	18%	(4.8, 3.9, 5.5)	12.2	0.23	16%	(4.8, 3.9, 5.5)	12.2	0.23	16%
	8	2	64	8.8	18	0.42	34%	(6.5, 5.7, 7.0)	18.3	0.48	30%	(5.2, 4.6, 6.0)	15.7	0.49	30%
	8	4	128	10.6	34.6	1.2	48%	(9.0, 7.5, 9.4)	34	1.3	44%	(6.9, 6.3, 7.8)	19.5	1.48	41%
	32	1	32	8.2	34	0.36	39%	(6.0, 4.7, 6.4)	30	0.39	37%	(6.0, 4.7, 6.4)	30	0.39	37%
	32	2	128	13.2	51	1.84	62%	(8.4, 8.0, 10.4)	49	1.93	60%	(7.7, 7.1, 8.8)	39.7	2.66	42%
	32	4	256	22	92	6.17	75%	(10.6, 8.8, 13.2)	100	6.32	73%	(8.0, 7.4, 8.9)	68.5	10.5	52%
	64	1	64	12	54	0.97	59%	(7.4, 5.1, 7.9)	54	1.0	57%	(7.4, 5.1, 7.9)	54	1.0	57%
	64	2	256	26	102	6.05	76%	(11.2, 9.2, 16.1)	105	6.21	74%	(8.3, 7.9, 9.4)	75	8.18	70%
2	32	2	128	16	54	4.03	57%	**bc***: (8.7, 5.8, 10.1) **wc***: (11.2, 9.2, 15.0)	59	4.43	52%	**bc**: (7.2, 5.0, 7.6) **wc**: (9.0, 6.2, 12.4)	42	4.61	50%
	64	2	256	24	145	13	71%	**bc**: (12.4, 10.5, 14.6) **wc**: (16.5, 14.0, 17.5)	124	13.94	66%	**bc**: (10.5, 9.0, 11.1) **wc**: (14.8, 13.5, 15.8)	87	17.42	53%
4	32	2	128	18	81	7.84	59%	**bc**: (9.0, 6.0, 12.0) **wc**: (11.9, 9.6, 15.0)	70	8.79	52%	**bc**: (7.8, 5.3, 8.0) **wc**: (9.9, 6.9, 14.3)	45	8.88	52%
	64	2	256	26	168	25.34	73%	**bc**: (13.9, 12.0, 15.0) **wc**: (17.9, 15.9, 18.8)	142	27.35	67%	**bc**: (11.5, 10.2, 12.4) **wc**: (15.5,14.0, 17.1)	101	31.84	58%
8**	32	4	128	–	–	–	–	**bc**: (13.9, 12.7, 16.5) **wc**: (15.5, 14.7, 19.9)	108	34.07	54%	**bc**: (12.8, 11.3, 14.2) **wc**: (13.4, 12.2, 16.4)	86	37.43	49%
16**	64	4	256	–	–	–	–	**bc**: (16.0, 13.6, 18.4) **wc**: (17.3, 16.2, 24.9)	240	126.39	85%	**bc**: (13.8, 12.0, 14.9) **wc**: (15.8, 13.9, 21)	187	252.78	58%

* **bc** : best-case and **wc**: worst-case cycle time based on address pattern
** 8-bank and 16-bank SRAMs are not supported by OpenRAM compiler

Table III compares the area, performance and power consumption for the same size of both asynchronous SRAMs. Fulcrum Microsystems uses dual-rail, 4 phase handshake and QDI timing model. As shown in this table for a 1Kx16-bit SRAM, AMC is 3.6X bigger in area. This can be attributed to two factors: (i) AMC uses a DRC-clean SRAM cell implemented with logic rules[2], while Fulcrum's memory uses TSMC's foundry cell that has significantly lower area; (ii) AMC uses four layers of metal to make the compiler portable for 65nm (for SCN3ME_SUBM 0.5μm technology only three layers of metals are used). This means data and address buses cannot be routed over the bitcells, and must be routed on the side. This can be seen in Figure 5, where buses are routed between the banks in both horizontal and vertical directions. Figure 7 from [5] appears to indicate that higher level busses are routed over the bitcells using additional metal layers, further reducing area. For a larger 16k $\times$ 64-bit SRAM array, Fulcrum's SRAM area is 1.466 mm^2[5] while AMC's is 3.34 mm^2—2.27X bigger, which is attributable to the difference in bit-cell area. We plan to incorporate a foundry cell into AMC in the future, which should reduce the area overhead of AMC-generated memory.

[2]We currently do not have access to the foundry bit cell.

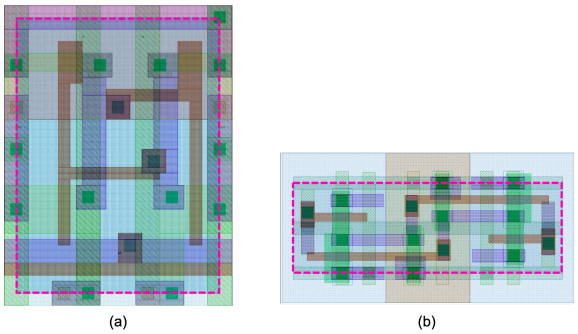

Fig. 8. 6-transistor (a) tall-cell layout in scn3me_subm and (b) thin-cell layout with no bends in polysilicon in 65nm. Dotted lines show the bounding box for each cell.

Fulcrum SRAM doesn't support a read-modify-write operation. The average frequency for its read and write operations is 1080 MHz while the average read and write frequency for AMC SRAM is 2156 MHz—a 2.0X improvement in performance. Cycle time breakdown is not showed by Fulcrum SRAM which makes it is hard to determine the source of longer cycle time for this SRAM. We speculate that employing sub-banking in AMC and using bundle-data encoding with extra timing assumptions in the AMC control logic are the main reasons for achieving higher performance. The SRAM

TABLE III
COMPARISON IN A $65nm$ PROCESS TECHNOLOGY

Fulcrum	**Conf.**	16 bank x (64 row x 16 col)
	Voltage	1.0 V
	Freq. (r, w)	(1023, 1137) MHz
	Avg. Power	7.5 mW
	latency	0.6 nS
	Area	0.019 mm^2 with foundry-cell
AMC	**Conf.**	4 bank x (4 sub-bank x (64 row x 16 col))
	Voltage	1.0V
	Freq. (r, w)	(2050, 2262) MHz
	Avg. Power	22.8 mW
	latency	0.29 nS
	Area	0.069 mm^2 with drawn-cell

generated by AMC consumes 3.0X power compared to same size Fulcrum SRAM. Again, the main source of this extra power consumption is manually drawn SRAM bitcell which is bigger in area compared to foundry cell and adds larger parasitics and hence increases the power consumption. For Fulcrum SRAM to operate at the same operation frequency as AMC SRAM (≈2 GHz), supply voltage must be increased to 1.5 V which results in 30 mW power consumption [5]. This increase in supply voltage to improve the performance leads to 20% more power consumption compared to AMC SRAM at the same operating frequency.

V. CONCLUSION AND FUTURE WORK

This paper introduced AMC: an open-source memory compiler that can be used by circuit designers and system architects. To our knowledge, this is the first attempt to make an open-source and portable asynchronous memory compiler. We believe making this compiler openly available has the potential to stimulate research in asynchronous design. Since the compiler can generate memories for real (as opposed to synthetic) process technologies, researchers can use AMC to build memories for their ASICs.

AMC generates DRC/LVS clean, fabricable GDSII layouts of variable-sized asynchronous SRAMs along with their SPICE, Verilog and timing/power models. AMC is a flexible compiler and can be quickly ported to different technology nodes. The SRAM generated by AMC: (i) uses the techniques in modern pipelined asynchronous designs to ease the timing constraints presented in clocked memories, (ii) provides a higher throughput and best-case behavior for latency, (iii) is pipelined allowing larger multi-banks SRAMs operate in a high frequency, and (iv) comes in different orientations and aspect ratio layouts with a tree bank structure.

AMC is an ongoing project. At the time of writing this paper, all the user-modifiable source code with reference implementations in SCMOS technology are released. This technology-independent compiler is ported to 0.5μm and 65nm technologies. We are planing to port the compiler to more scaled technologies including 28nm, 14nm (FinFET), and incorporating foundry bit-cell layout. Also, we are adding the option for using higher metal layers, which has the potential to improve the area and performance of memory. In addition, system-level implications such as error correction code (ECC), soft-error redundancy and build-in self test (BIST) are on the future-work list. There are also some additional circuit optimizations that are possible, including better sizing and handshake optimizations. Our goal is to work with the community to provide a fully open-source, high-quality memory compiler beneficial for research, education and making real products. AMC enables rapid prototyping for researchers in various fields from computer architecture to SoC design, device research, and computer-aided design.

REFERENCES

[1] R. Goldman, K. Bartleson, T. Wood, V. Melikyan, and E. Babayan, "Synopsys educational generic memory compiler," in *10th European Workshop on Microelectronics Education (EWME)*, p. 8992, May 2014.

[2] M. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "OpenRAM: An open-source memory compiler," in *IEEE International Conference On Computer Aided Design (ICCAD)*, November 2016.

[3] A. J. Martin, A. Lines, R. Manohar, M. Nystroem, P. Penzes, R. Southworth, and U. Cummings, "The design of an asynchronous MIPS R3000 microprocessor," in *Advanced Research in VLSI*, pp. 164–181, - 1997.

[4] M. Wieckowski, "GDS Mill user manual." http://michaelwieckowski.com/wp-content/uploads/2012/01/GdsMillUserManual.pdf, 2012.

[5] J. Dama and A. Lines, "Ghz asynchronous SRAM in 65nm," in *15th IEEE Symposium on Asynchronous Circuits and Systems*, p. 8594, May 2009.

[6] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Sixth MIT Conference on Advanced Research in VLSI* (W. J. Dally, ed.), pp. 263–278, 1990.

[7] R. Manohar and Y. Moses, "Analyzing isochronic forks with potential causality," in *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*, pp. 69–76, IEEE, 2015.

[8] V. N. Ekanayake and R. Manohar, "Asynchronous dram design and synthesis," in *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, pp. 174–183, IEEE, 2003.

[9] C. Kelly IV, V. Ekanayake, and R. Manohar, "SNAP: A sensor-network asynchronous processor," in *Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*, pp. 24–33, IEEE, 2003.

[10] C. T. O. Otero, J. Tse, R. Karmazin, B. Hill, and R. Manohar, "ULSNAP: An ultra-low power event-driven microcontroller for sensor network nodes," in *15th International Symposium on Quality Electronic Design*, pp. 667–674, IEEE, 2014.

[11] MOSIS, "MOSIS scalable CMOS (SCMOS)." https://www.mosis.com/files/scmos/scmos.pdf, 2018.

APPENDIX

Following is the notation we use for the CHP language.:

- **Assignment**: $x := E$. This statement means "assign the value of expression E to x."
- **Communication**: $A!E$ is a statement meaning "send the value of expression E over channel A," and $B?x$ means "receive a value over channel B and store it in variable x." Both sending and receiving are blocking.
- **Choice**: $[G_1 \rightarrow P_1[] \ldots []G_n \rightarrow P_n]$, where each G_i is a Boolean expression (guard), and each P_i is a program fragment. This statement is executed by waiting for exactly one of the guards to be true, and then executing the associated fragment.
- **Repetition**: $*[P]$ infinitely repeats statement P.
- **Sequential Composition**: $P; Q$.
- **Parallel Composition**: P, Q.