# Hierarchical Instruction Register Organization

David Black-Schaffer, James Balfour, William J. Dally, *Fellow*, Vishal Parikh, JongSoo Park
Computer Systems Laboratory, Stanford University, Stanford, CA, USA
{davidbbs, jbalfour, dally, vparikh1, jongsoo}@cva.stanford.edu

***Abstract*— This paper analyzes a range of architectures for efficient delivery of VLIW instructions for embedded media kernels. The analysis takes an efficient Filter Cache as a baseline and examines the benefits from 1) removing the tag overhead, 2) distributing the storage, 3) adding indirection, 4) adding efficient NOP generation, and 5) sharing instruction memory. The result is a hierarchical instruction register organization that provides a 56% energy and 40% area savings over an already efficient Filter Cache.**

***Index Terms*— energy-efficient embedded processor architecture, hierarchical and distributed instruction register organization, VLIW instruction delivery**

## I. Introduction

INSTRUCTION delivery in embedded processors consumes up to 30% [12] of total energy due to the use of instruction caches. These caches are generally sized to be as large as possible and still allow single-cycle access. As memory power scales with the size of the memory, this sizing results in caches that may consume significantly more power than necessary.

This work focuses on the sizing and organization of the instruction memories so as to deliver instructions to the functional units of a VLIW processor efficiently across a range of embedded media kernels. The kernels examined consist of tight processing loops of fewer than 100 5-wide VLIW instructions each. Longer programs are not considered for two reasons: 1) the execution of embedded media applications is dominated by small kernels such as those considered here, and 2) the execution of longer blocks of code necessarily requires accessing the much larger memory that contains them, which results in the issue energy being dominated entirely by this larger memory, regardless of what is done at the micro-architectural level. Therefore, to achieve high instruction issue efficiency for embedded media applications, it is necessary to deliver instructions from these small loops to the functional units with as little energy as possible.

The architectures evaluated in this paper use small register files (Instruction Register Files, or IRFs) to issue the instructions[1]. For the VLIW processor considered here, there are many options for the sizing and distribution of these IRFs due to the natural instruction fields contained in the VLIW instruction word. This work starts with a baseline cache implementation of the same size as the IRFs and compares it to a variety of organizations. The analysis indicates that removing the tags saves 5% of the energy, while distributing the memories and indexing the issuing can save 20% and 21% more, respectively. Adding efficient NOP generation, which effectively allows the issue width to be dynamically adjusted, saves an additional 20%, and sharing the instruction memories between functional units can garner an additional 13%. Overall, the hierarchical IRF approach suggested here results in an energy savings of 56% compared to an already efficient Filter Cache [6].

Manuscript submitted: 27-Mar-2008. Manuscript accepted: 28-Apr-2008. Final manuscript received: 30-Apr-2008. This work was funded by SRC Contract 2007-HJ-1591, Intel, and the Stanford Center for Integrated Systems.

[1]The IRFs presented here are significantly different from previous IRFs [3] in that code is executed directly from them, rather than using them purely as a lookup table for frequent instructions.

## II. Instruction Register File Architecture

The Instruction Register File architecture consists of a Control and Index Memory (CIM) and one or more possibly distributed Instruction Register Files (IRFs), which are integrated into the functional units they control if they are distributed. (See Figure 1.) The CIM is addressed directly by the current Program Counter (PC) and the IRFs are addressed by indices from the CIM. Each entry in the CIM corresponds to one full VLIW instruction, and is logically divided into two portions: control and index. The control portion of the CIM contains the control flow (e.g., branch, jump) instructions for the processor. The index portion of the CIM contains the indices (e.g., addresses for the IRFs) of the instructions to be issued from each IRF for each cycle. This structure allows for a level of indirection whereby multiple VLIW instructions (entries in the CIM) can reuse functional unit control bits in the IRFs, thereby efficiently storing repeated instructions. Such an indirection requires that the memories be fast enough that both the CIM and the IRFs can be accessed in one cycle, which is realistic for the selected memory sizes in this technology.

The IRF architecture defines a unique name space for the loaded instructions, with all execution and control flow taking place within the IRF. Unlike a cache, the IRF does not maintain any tag bits to associate loaded instructions with addresses in main memory. This reduces the size of the memory (by eliminating tags) and the complexity of the PC update logic (by reducing its size), at the expense of requiring software-coordinated instruction loads.

### A. Loading Instructions into IRFs

Instructions are loaded in blocks to amortize the overhead of loading and to store the instructions efficiently higher up in the memory hierarchy. A hardware engine, the Instruction Load Engine (ILE), is responsible for performing the transfers from memory to the CIM and IRFs when a "load instruction block" instruction is executed. Each block contains a header that specifies how many instructions should be loaded into each IRF. The ILE enables the loading of instructions while execution continues through the use of a presence bit for each CIM entry.

To improve efficiency, two optimizations are applied to each block of code as it is loaded. For embedded media applications with significant temporal code reuse, the overhead of applying these optimizations at load time is significantly cheaper than doing so dynamically each time the instructions are executed. The first optimization is to re-write the addresses of relative

Posted to the IEEE & CSDL on 5/30/2008
DOI 10.1109/L-CA.2008.7 1556-6056/08/$25.00 © 2008 Published by the IEEE Computer Society

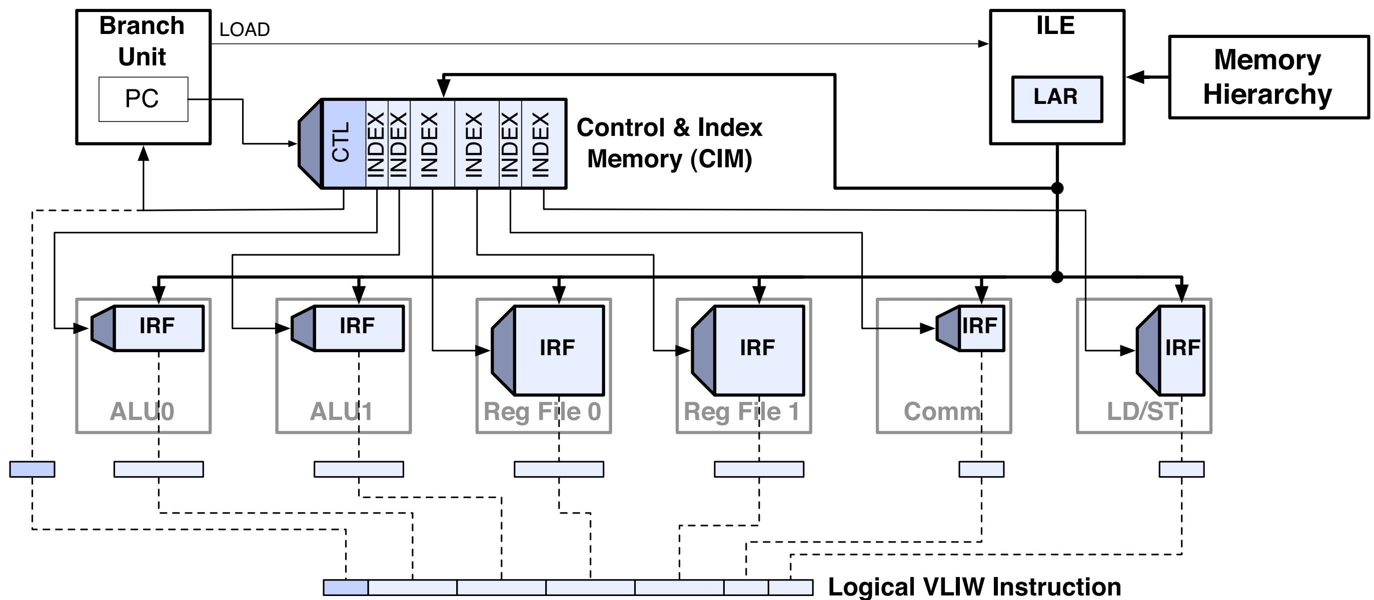


Fig. 1. The IRF architecture. The full VLIW instruction logically consists of the IRF outputs and the control bits from the CIM, as illustrated at the bottom of the figure. The actual instruction bits are kept local to the unit that consumes them, as is indicated by the separate IRFs for each functional unit. IRFs can be shared between multiple functional units by using multi-ported registers, as in Figure 2f.

branch instructions as they are loaded. In traditional architectures, a relative branch is calculated by adding an offset to the PC every time the branch is evaluated. By doing this calculation once while loading the instruction block, however, it is no longer necessary to re-calculate the destination on every evaluation of the branch.

The second load optimization is to store the addresses for the instruction block loads in a side structure, the Load Address Register (LAR), rather than encoding them in the instruction itself. Keeping the addresses in the LAR reduces the number of bits required to specify each “load instruction block” instruction considerably, as it need contain only a reference to the particular LAR to access for the address. The LAR is loaded from the instruction block by the ILE.

## III. Evaluation

The IRF architecture was evaluated with 7 benchmark kernels (128-bit AES, 3×3 2D convolution, crc32, 8×8 DCT, 32-tap 16-bit FIR, JPEG DC Huffman encoding, and GSM Viterbi with 189-bit frames), ranging from 22 to 73 instructions and 140 to 18,073 cycles. The target processor was a 5-wide VLIW machine with the functional units shown in Figure 1. The analysis was accomplished by running the benchmarks on a RTL-level simulation of the architecture and extracting instruction traces. The traces were then analyzed to determine the required sizes for the IRFs for each configuration. The access counts were then determined by simulating the execution of each trace on each configuration. Memory and wire energy were determined from standard VLSI models for SRAMs in a 130nm process at 1.2V, including appropriately sized address decoders, bit-line read and write circuits, cell, and bit- and word-line loads, all with appropriate activity factors[2]. Leakage was deemed to be insignificant in this process with such small memories with high activity factors. The relative sizing was determined from a fully placed-and-routed version of the RTL. The wires distributing data from centralized structures were assumed to travel half way across the long axis of the placed-and-routed design[3].

The baseline design point was the Filter Cache, Figure 2a. The Filter Cache was sized to be similar to the total size of the IRFs (64 entries deep × 135 bits for the instruction plus 10 bits for the tag) to provide a purely architectural comparison of the overhead of the dynamic tag array. The energy breakdown of the cache (Figure 3a) indicates that 6% of the energy was spent in the tag array and 26% in the wires distributing the instructions from the centralized memory to the functional units. The tag energy is such a low percentage due to the small address space considered ($2^{16}$ instructions) and the relatively large instruction word (135 bits vs. 10 for the tag). Compared to the cache, the Monolithic Full configuration (Figure 2b) removes only the dynamically configured tag bits. This configuration is roughly the same as a pre-loaded loop cache [7]. For this configuration the CIM is extended to contain the full instruction word for each entry. The instructions are still distributed from a centralized monolithic CIM memory so the wire energy will be the same as the cache example. The memory array for the instructions is the same size as the array for the cache minus the tags. The evaluation of the Cache vs. the Monolithic Full configuration demonstrates that a software-controlled memory can save the tag energy compared to a cache, but suffers from the same large wire overhead due to its centralized nature.

To analyze the cost of a centralized memory, the Distributed Full configuration (Figure 2c) distributes the IRFs for each functional unit, but does not add indexing. The CIM in this configuration controls only sequencing, with the program counter directly controlling each IRF. This implies that all IRFs are the same depth

[2]The overhead for multiple small memories is mostly due to the initial decoder and the pre-decode lines, which is small as the energy is dominated by the actual bit-line accesses. This is typically not the case with RAM compilers which use a “one size fits all” approach, resulting in over-designed peripheral circuitry for small memories.

[3]The impact of the wire energy in this model is directly proportional to this assumption.

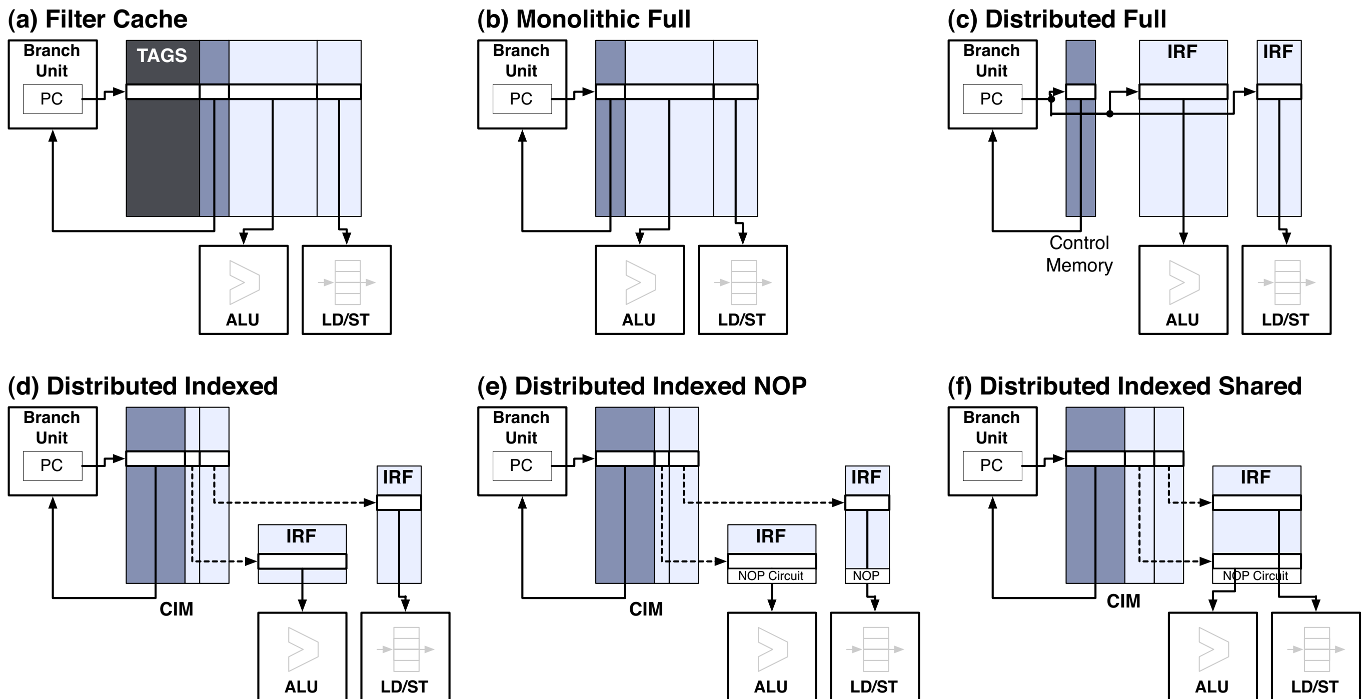


Fig. 2. IRF configurations evaluated. Only two functional units are shown for simplicity. For $a$ and $b$, the memories are centralized so the wire cost is that of sending the instruction bits from the centralized memory to the functional units. For $c - f$, the memories are distributed, so the wire cost is that of sending the addresses from the CIM indices to the distributed IRFs.

as the CIM as they are all indexed by the PC. Compared to the Monolithic Full design, the total energy decreases by 20%, while the energy spent on wires decreases by 90%. The energy spent in the memory actually increases by 12% in this configuration due to having added separate address decoders for each of the distributed IRFs, while the Monolithic Full configuration has the same number of bits but only one decoder for the single memory. The area also increases proportionally due to the decoders as can be seen in Figure 3b. The wire energy is decreased by a factor of ten since the Distributed Full configuration need only distribute the PC from the centralized branch unit to each of the IRFs, rather than distributing the full instruction word from a centralized memory to each of the functional units. This indicates that the major benefit of distributed structures is in more efficiently encoding the information sent over long wires, but that this must be balanced with the overhead of the additional peripheral circuitry required to operate multiple memories.

The Distributed Indexed configuration (Figure 2d) adds indirection from the CIM indices to the IRFs. This indirection allows the individual IRFs to be resized to be only as large as needed, and results in an energy reduction of 21% compared to the Distributed Full configuration. The area spent on the CIM increases by 2.5× as the Distributed Indexed configuration requires indices in the CIM to address the IRFs. However, this increase is more than compensated for by the IRFs which decrease by 52%, leading to a net area decrease of 29%. The energy spent on the wires does not change because the number of bit transitions for each IRF index is the same. The difference is that the memories are only as large as needed, which reduces the read energy. Overall, adding the indirection from the CIM to the IRFs allows the IRFs to be reduced in size, resulting in an energy reduction of 21% compared to the Distributed Full configuration and 37% compared to the Filter Cache, despite the addition of the index bits in the CIM. The reduction in the size of the IRFs does not result in a wire energy savings because the same number of index bit transitions are required to execute the same program as the packing is no different from the Distributed Full configuration.

The Distributed Indexed NOP configuration (Figure 2e) adds hardware NOP generation to the IRFs. This eliminates the need to read NOP instructions out of the IRF and instead generates them with a simple circuit which is roughly 10 times as efficient. This addition reduces the energy by 20% at an area cost of 3%, indicating that NOPs can be very efficiently encoded if the instructions can be efficiently distributed. In a centralized instruction delivery architecture, the energy spent distributing the NOPs over wires would be much larger, reducing the benefits from generating them efficiently. Here the distribution overhead is assumed to be negligible (the IRFs and the NOP generation circuits are located within the unit which they control) so the architecture realizes the full benefit of efficiently generating the NOP.

The final configurations (Shared A and B, Figure 2f) evaluate various ways of sharing IRFs between functional units. The combinations evaluated include various choices for sharing the instruction bits for the ALU op codes, ALU source/destination control bits, memory load/store unit, and communications unit. Five sharing combinations were analyzed, but only the two with the best energy and area are presented here. Both shared configurations place the read and write control bits for both register files in one shared IRF and share the CIM between the control, load/store, and communications units. Configuration A additionally places all control for both ALUs in one shared IRF. For the shared configurations, energy savings was at most 13% compared to the Distributed Indexed NOP (Shared B) and the area

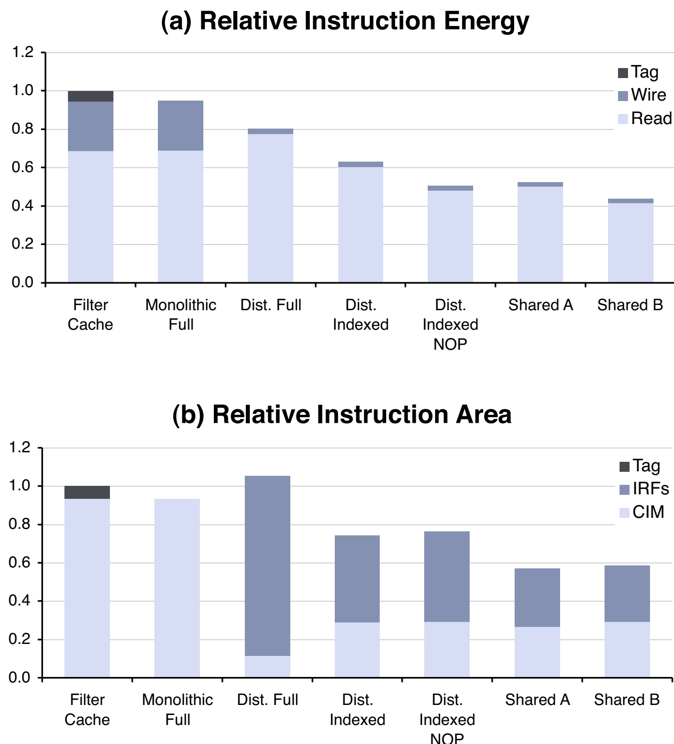


Fig. 3. IRF Energy and Area across all benchmarks, normalized to the Filter Cache baseline.

reduction was at most 32% (Shared A). The benefits of sharing the IRFs is limited because the number of instruction bits read out in both the shared and non-shared cases is the same, as they execute the same program, and therefore the read energy will be equivalent. The 32% reduction in area does not translate into an energy savings in this model as the memories are assumed to be positioned between the shared units and therefore incur no additional communications costs on the reads. This reduction in area could be significant depending on the ratio of logic area to instruction storage area in the overall processor.

## IV. Related Work

Instruction power reduction for VLIW processors has been extensively studied due to the large width of the instructions and the frequency of NOPs for various functional units [4], [8]. Approaches have included instruction compression and scheduling to minimize the instruction bus activity factor [5]. The indexed IRF architecture effectively implements compressed instruction loads, but could potentially benefit from wire activity-aware scheduling.

Filter [6] and Loop Caches [7] have long been presented as energy-efficient instruction stores for embedded processors with tight loops, and correspond to the Filter Cache and Monolithic Full configurations. These two approaches have been shown to reduce instruction fetch power by 58% and 60% relative to a standard I-cache, respectively [6], [2]. Clustered loop buffers [5] were proposed to build an efficient and scalable loop buffer for an 8-wide VLIW processor, achieving 63% energy savings. Their implementation is similar to the Distributed Indexed configuration described here, but their distributed control resulted in worse efficiency, possibly due to disregarding the wire energy.

Previous work on Instruction Registers [3] focused on improving efficiency by using the IRF as a compressed instruction storage and not executing directly from it. Their approach reduced the energy required to issue compared to a standard I-cache by 37%, which is less than the effect of our baseline Filter Cache [6]. Microcode and microprogramming [1], [13] have a long history as techniques to compress instruction streams. Two-level control stores for the microprogammable Warp cell were proposed to reduce code storage pressure [9]. The Warp structure resembles the indirect indexing of the IRF architecture proposed here, with their microstore and nanostore operating much as the CIM and IRFs. Block instruction load formats have been used to amortize instruction fetch in architectures such as TRIPS [11], [10].

## V. Conclusions

This work has evaluated a range of configurations for efficient instruction delivery for the critical portions of embedded media applications on a VLIW processor. The evaluation included the cost of the memory access and wires to distribute the instruction bits. The most efficient configuration distributes, indexes, and shares individual memories for the functional units, and uses custom circuits to efficiently generate NOPs, resulting in a 56% decrease in energy and 40% decrease in area compared to a baseline Filter Cache. The benefits of re-writing instructions as they are loaded were also discussed.

## References


[1] M. J. Flynn and M. D. McLaren, "Microprogramming revisited," in *Proceedings of the 1967 22nd national conference*. New York, NY, USA: ACM, 1967, pp. 457–464.

[2] A. Gordon-Ross, S. Cotterell, and F. Vahid, "Tiny instruction caches for low power embedded systems," *Transactions on Embedded Computer Systems*, vol. 2, no. 4, pp. 449–481, 2003.

[3] S. Hines, G. Tyson, and D. Whalley, "Reducing instruction fetch cost by packing instructions into register windows," in *Prof. of ACM/IEEE International Symposium on Microarchitecture*, 2005.

[4] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir, "Introducing the IA-64 architecture," *IEEE Micro*, vol. 20, no. 5, pp. 12–23, 2000.

[5] M. Jayapala, F. Barat, T. Vander Aa, F. Catthoor, H. Corporaal, and G. Deconinck, "Clustered loop buffer organization for low energy VLIW embedded processors," *Transactions on Computers*, vol. 54, no. 6, pp. 672–683, 2005.

[6] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The Filter Cache: an energy efficient memory structure," in *International Symposium on Microarchitecture*, 1997.

[7] L. Lee, B. Moyer, and J. Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," in *International Symposium on Low Power Electrionics and Design*, 1999.

[8] J. Liu, B. Bell, and T. Truong, "Analysis and characterization of Intel Itanium instruction bundles for improving VLIW processor performance," *Computer and Computational Sciences, 2006. IMSCCS '06. First International Multi-Symposiums on*, vol. 1, pp. 389–396, 2006.

[9] O. Menzilcioglu, "A case study in using two-level control stores," *SIGMICRO Newsl.*, vol. 19, no. 3, pp. 46–48, 1988.

[10] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler, "A design space evaluation of grid processor architectures," in *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 40–51.

[11] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger, "Distributed microarchitectural protocols in the TRIPS prototype processor," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 480–491.

[12] S. Segars, "Low power design techniques for microprocessors," in *Tutorial, International Solid State Circuits Conference*, February 2001.

[13] M. Wilkes, "The best way to design an automatic calculating machine," *Manchester University Computer inaugural Conference*, pp. 16–18, July 1951.