TTAs: Missing the ILP complexity wall

Henk Corporaal

Delft University of Technology
Department of Electrical Engineering
Mekelweg 4, 2628 CD Delft, The Netherlands

Abstract

A common approach to enhance the performance of processors is to increase the number of function units which operate concurrently. We observe this development in all recent general purpose superscalar processors, and in VLIW (very long instruction word) processors used for more dedicated application domains, like the multi-media domain.

This paper analyzes the data path complexity of ILP processors (in particular VLIWs), and shows that they soon may hit the complexity wall; their complexity gets out of control when scaling to very high performance. Several methods are investigated for reducing this complexity. Essentially these methods trade hardware for software complexity, i.e., performing as much as possible at compile time. Combining these methods results in a new architecture, called transport triggered architecture or TTA.

The concept of transport triggering is outlined together with its characteristics. It will be shown that the application of this concept results in a number of hardware advantages, and introduces a number of new scheduling optimizations. Together they substantially reduce the ILP complexity bottleneck, which will be demonstrated by a number of experiments.

Keywords: instruction level parallelism, architecture complexity, VLIW, Superscalar, bypassing

1 Introduction

In order to fulfil continuously increasing demands for more processing power, most processor manufacturers upgraded their general purpose processor architectures with superscalar capabilities. This allows them to stay binary compatible with what is already on the market. However, as is well known, the exploitation of instruction-level parallelism using superscalar techniques is rather limited. Primarily this is caused by the limited instruction window of the hardware instruction dispatch mechanism. Enlarging this window results in high hardware costs for dependence checking and resource allocation, extra pipeline stages (which increase branch penalties), and a possible increase of the achievable cycle time [1].

A different, but binary incompatible, approach to exploiting instruction level parallelism (ILP) is taken by VLIW architectures. Currently, several interesting VLIWs hit the market, like the Trimedia of Philips, the Mpact of Chromatic and the TMS320C6x of Texas Instruments. They deliver high performance at reduced cost. The superscalar hardware has been replaced by compile-time dependence checking and resource allocation. Despite their good properties, the data path of VLIWs is still too complex, in particular when they are scaled to very high performance. This makes it interesting to look at alternative architectures which avoid this complexity, but keep the good properties of VLIWs.

In this paper we analyze the data path complexity of VLIW processors. We demonstrate several methods to reduce this complexity by replacing run-time complexity by compile-time complexity. This results in a new architectural concept: the concept of *transport triggering*. Transport triggered architectures (TTAs) are based on this concept. It will be demonstrated that TTAs do not hit the ILP complexity wall.

This paper is structured as follows. Section 2 classifies different types of ILP architectures with respect to the amount of hardware they contain for parallelism detection. Section 3 introduces the required terminology concerning the complexity of the data path of ILP processors and applies this terminology to RISC processors. Section 4 treats the complexity of VLIW processors. Section 5 demonstrates several methods to reduce ILP processor data path complexity. Section 6 describes an approach we have taken to reduce complexity: transport triggered architectures are described. A quantitative evaluation of three main TTA advantages is presented in section 7. Finally, section 8 presents a summary and draws major conclusions.

2 ILP architectures and the role of the compiler

Exploiting parallelism has its price: the parallelism has to be detected and exploited efficiently by the architecture under consideration. Let us therefore look at which translation and interpretation steps have to be taken in order to execute a sequential program written in a HLL on a single ILP processor:

1. Frontend compilation. Lexical analysis and parsing of the program, performing optimizations, and compilation to basic operations. During this step alias analysis is also performed.
2. Determine dependences and produce data and control dependence graphs.
3. Bind operands to locations. E.g., non aliased variables and temporaries are bound to local registers for a certain life-time in order to reduce off-chip data traffic; this is called *register allocation*.
4. Bind operations to time slots, that is, decide when to execute operations and data transports. This is usually called *scheduling*.
5. Bind operations to FUs. When there are multiple function units of the same type, a binding choice has to be made.
6. Bind transports to buses. Performing operations requires the transport of many data values; buses must be allocated to these transports.
7. Execute the operations and perform the transports.

The order of these steps is not necessarily the same as presented above. It is clear that the first step is performed by the compiler, and the final step by the hardware. Intermediate steps may either be performed by hardware or the compiler. This is shown in figure 1. It illustrates how this division is made for different architectures.

Superscalar architectures shift most responsibilities to the hardware. Although they are able to run unmodified object code of former generation sequential architectures, they certainly need compiler assistance to execute programs efficiently. Hardware complexity limits superscalars to a very small run-time scheduling window (see e.g. [1]); therefore the compiler has to schedule code in such a way that operations within the scheduling window are independent from each other as much as possible.

Dataflow architectures let the compiler perform the frontend compilation and the data dependence analysis; other steps are performed by hardware. Dependences are made explicit in the object code, that is, each

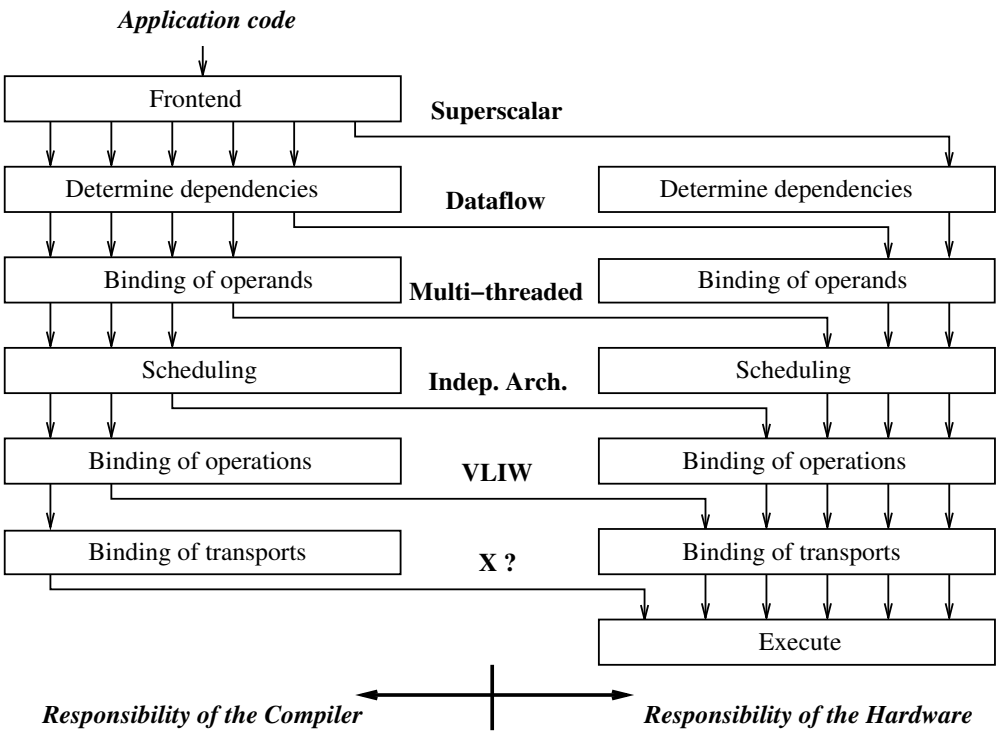


Figure 1: The division of responsibilities between hardware and compiler.

operation specifies its successor operations to which the result value(s) of the operation must be send. These values are sent by so-called *tokens*; they contain both the value and the successor specification. When tokens are received they have to be matched at run-time with instructions and other tokens (most instructions require multiple tokens). This matching turns out to be very expensive and time consuming [2].

Multi-threaded architectures can be considered as architectures which do the scheduling at run-time, but, in contrast to dataflow architectures, rely on the compiler for binding of operands to locations (this avoids token matching). A further difference concerns the scheduling granularity; it is not one instruction but a thread, containing multiple instructions. Although multi-threaded architectures support *fine grain* execution in the form of small threads, the compiler should try to enlarge threads (and therefore the scheduling granularity) in order to avoid the still existing thread switching overhead. A number of multi-threaded architectures have been proposed. These architectures support multiple threads by offering hardware support for efficient thread synchronization and thread switching. Just like superscalar and dataflow architectures, they can tolerate long, possibly non-deterministic, latency operations, like data loads, by switching quickly to another runnable thread of execution. Some of these architectures are presented as descendents from dataflow architectures [3, 4]. They tend to avoid the dynamic token matching. Others are inspired by concurrent object oriented computation, like the J-machine [5] and Sparcle [6].

More recently several research groups have proposed usage of multi-processor architectures running multiple threads extracted from a single (sequential) program, with the aim to exceed the amount of instruction level parallelism exploitable from a single thread [7, 8, 9, 10, 11]. Each thread is run on a separate processor, presumably a superscalar processor; this way they combine both the exploitation of fine grain (at the instruction level) and more coarse grain (thread level) parallelism.

Independence architectures require that the compiler has to specify independent operations whose operands are available in a given cycle; in other words, the compiler performs the scheduling. The hardware performs the binding of operations and transports to hardware resources. An example is the Horizon architecture as presented in [12]; each operation encodes a number H which specifies the next H concurrent operations. The hardware can execute these H (or less) operations concurrently, without having to test if they are dependent or whether their operands are ready. The hardware still has to allocate (bind) these operations to FUs.

VLIW architectures take a more radical approach; apart from the binding of transports to buses they perform all steps in software. This results in a simple and cheap architecture. However, as will be shown the data path of VLIWs is still too complex when scaled to very high performance levels. One may therefore ask the question if it makes sense to perform the final step also in software as well: what kind of architecture results and what are its characteristics? Before answering that question we will perform a detailed research of the data path complexity of VLIW processors.

3 Data path complexity: terminology

In general an instruction specifies one or more operations, and for each operation zero or more source operands and also zero or more destination operands. These operands are accessed either from the register file (RF) or from memory. The latter may require complex address arithmetic. Small immediates are directly accessible from the instruction register. The next discussion about data path complexity concentrates on the transport of these operands. We start with the data path of a non-pipelined processor, and pipeline it afterwards.

In a non-pipelined processor the instruction fetch of the next instruction has to wait until the write-back of the current instruction has been completed. The data path of a (very simple) non-pipelined processor is shown in figure 2a. It contains a general purpose RF for the operand and result values, and one function unit (FU) performing all the required operations as specified by the instruction set of this processor. The FU has two inputs and one output. Note that one operand register and the result register could be omitted, however they facilitate ease of design and implementation.

To enable all required data transports a certain connectivity between source and destination locations of operands is required. Figure 2b shows the connectivity graph for the non-pipelined processor. A connectivity graph has a related architectural complexity.

Definition 3.1 *The* architectural connectivity complexity, AC_{compl}, *of a connectivity graph* $CG(S, D, E)$, *is defined as a 3-tuple,* $(\#S, \#D, \#E)$, *where* $\#Sn$ *is the number of source nodes, or* $\#Sn = |Sn|$, $\#Dn$ *is the number of destination nodes, and* $\#E$ *is the number of edges contained in the graph* CG.

For the given graph the architectural complexity is

$$AC_{compl}(\text{non-pipelined}) = (N + 5, N + 5, 3N + 4) \tag{1}$$

with N equal to the number of general purpose registers.

Although the connectivity graph shows which sources and destinations need direct connections, it does not tell how to implement this connectivity; many options for the data path exist, ranging from a single

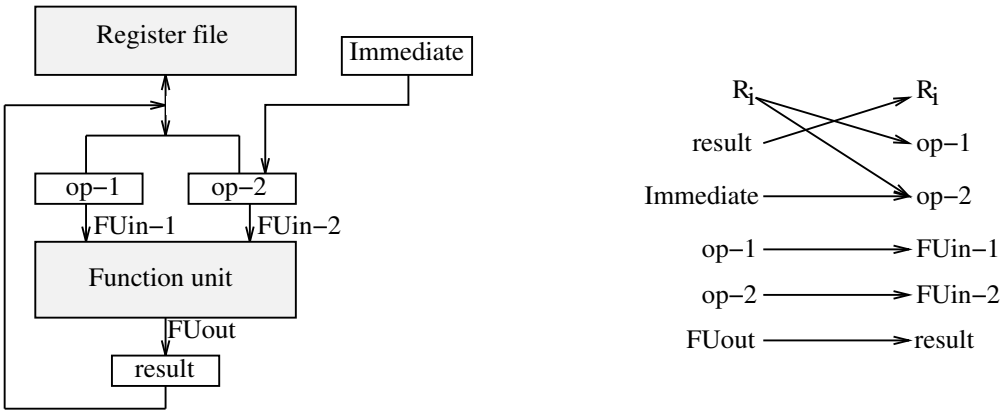


Figure 2: Possible data path and required connectivity of a non-pipelined processor.

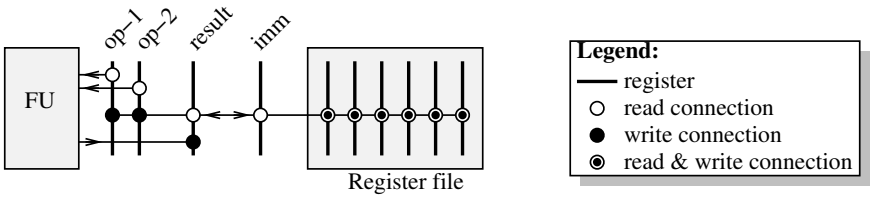


Figure 3: Connectivity model of a non-pipelined processor.

bus connecting all sources and destinations, to separate connections for each edge in CG. Its architectural complexity is therefore an indication, but not a real measure, of the implementation complexity. We need therefore to define the data path complexity.

Definition 3.2 *The* data path complexity, DP_{compl}, *of a processor data path is defined as a 5-tuple,* $(\#Bus, \#RC, \#WC, \#RP, \#Regs)$, *where* $\#Bus$ *is the number of data buses,* $\#RC$ *the number of read connections in the data path,* $\#WC$ *the number of write connections in the data path,* $\#RP$ *the number of ports on the RF, and* $\#Regs$ *the total number of registers required, including general purpose registers and registers to hold immediates, but exclusive internal FU registers.*

Besides the data path complexity, which gives a quick indication of the complexity of the communication requirements, we need measures for the timing criticalness of the data path. The achievable cycle time of a data path is largely determined by two factors: the number of read connections to a bus and the number of write ports to a register highly determine the read and write delays [13, 14]. We therefore define:

Definition 3.3 RC_{max}, *the maximum number of read connections to any bus, and* WP_{max}, *the maximum number of write ports to any register.*

An implementation of any connectivity graph requires at least one shared read-write bus. This results in a non-pipelined processor, because a maximum of only one data transport per cycle is supported. The data path of figure 2a shows such an implementation. The data path complexity for this non-pipelined processor can be determined from figure 3; this figure shows a different view of the data path, including all

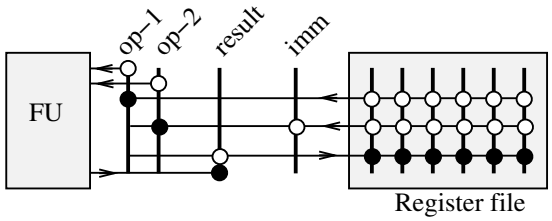


Figure 4: Connectivity model of a pipelined processor.

the necessary read and write connections; we call it the *connectivity model* of the processor. The data path complexity for this non-pipelined processor is given by:

$$DP_{compl}(\text{non-pipelined}) = (4, N+4, N+3, 1, N+4) \tag{2}$$

Further, we count that $RC_{max} = N+2$, and $WP_{max} = 1$; there is no register with more than one write port. A few remarks are appropriate at this point:

- The listed read and write connections do not include internal FU connections; their numbers are highly dependent on the implementation of the FU. We further do not include the write connection(s) to the immediate register.
- Of the four buses three are very simple; they only serve to connect FU in and outputs to corresponding source operand and result registers.
- If there is only a single read connection to a bus, it does not require read select logic; similarly a single write connection to a register does not need write select logic. These connections are therefore very cheap. In order to simplify matters, we do not distinguish them from other connections.

What happens to the data path if we want to pipeline this processor such that we can issue a new instruction each cycle? For the data path this means that we have to add extra buses: the source operands and the FU result value have to be read concurrently from and written to the RF. Figure 4 shows the connectivity model.

Although its architectural connectivity complexity is exactly the same as for the non-pipelined version (the connectivity graphs are identical), its data path complexity, DP_{compl}, has increased to:

$$DP_{compl}(\text{pipelined}) = (6, 2N+4, N+3, 3, N+4) \tag{3}$$

and therefore, compared to the non-pipelined version, pipelining contributes the following to DP_{compl}:

$$\Delta DP_{compl}(\text{pipelined,non-pipelined}) = (2, N, 0, 2, 0) \tag{4}$$

In this case $RC_{max} = N+1$, and $WP_{max} = 1$, as before. So the primary difference in the complexity of a pipelined processor is the addition of extra buses, and the corresponding register ports.

A fundamental performance bottleneck results when pipelining processors: results of FUs have to be forwarded in order to get single cycle instruction throughput [15]. Forwarding for a 4-stage pipelined RISC processor is indicated in figure 5. Figure 6 shows the connectivity graph, and a possible data path of this processor, including the bypass circuit.

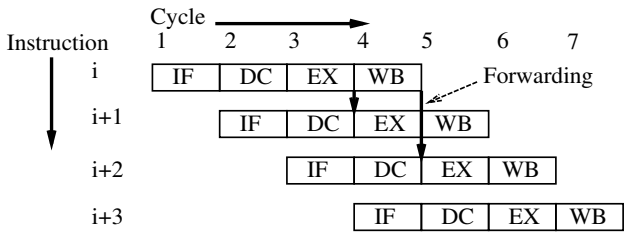
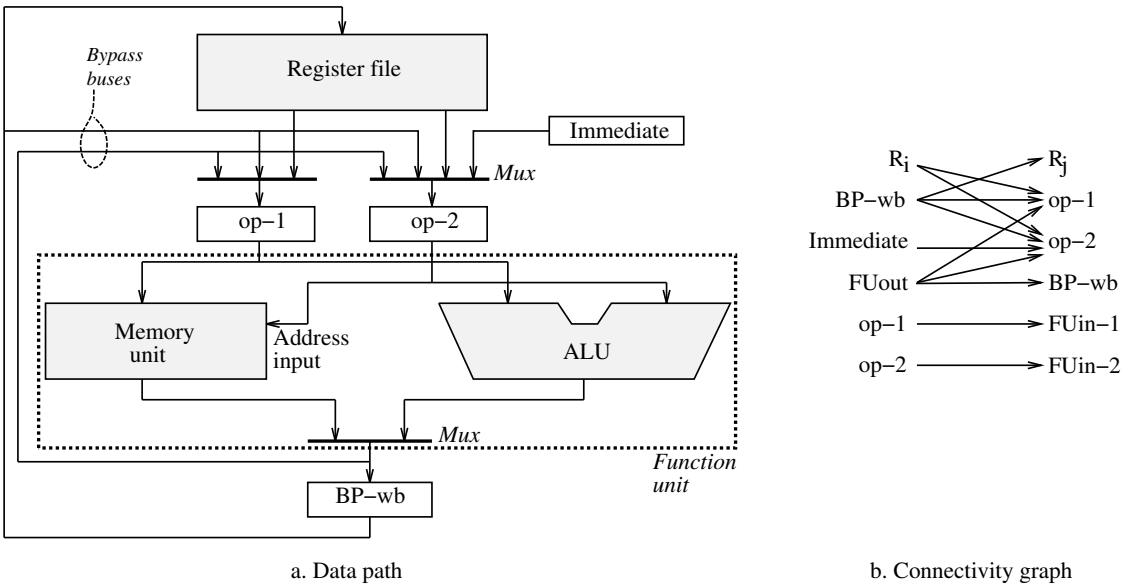


Figure 5: RISC pipelining diagram (4-stage pipeline).



a. Data path

b. Connectivity graph

Figure 6: RISC data path and connectivity graph.

The function unit (FU) contains an ALU (arithmetic and logical unit) and a memory unit. The FU is responsible for executing all arithmetic, logic and memory operations, including loads and stores. In order to execute (data) memory accesses in the same time as an ALU operation the memory includes a single cycle access data cache (mostly realized on-chip). The FU has two inputs connected to the registers op-1 and op-2, which can be fed by data from the RF, or from the bypass circuit. To allow for immediate operands coming directly from the instruction stream, op-2 has an extra input for immediates. The BP-wb (bypass write-back) register holds the result value for one cycle.

Bypassing changes the architectural complexity, AC_{compl}, of the connectivity graph (figure 6b); for this RISC processor it is equal to:

$$AC_{compl}(\text{RISC}) = (N+5, N+5, 3N+8) \tag{5}$$

That means, bypassing adds four connections, but does not change the number of source and destination nodes (compare equation 1). The corresponding connectivity model of this RISC processor is shown in figure 7. In this figure we distinguish operand, bypass, and other registers.

The *bypass circuit* is defined to contain the source operand registers, the bypass registers (in this case only BP-wb), and their connectivity. Note that in order to select the right operands for the FU the source-ids

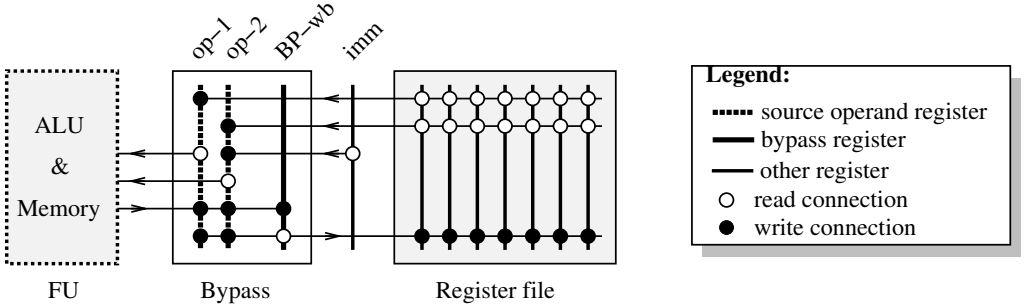


Figure 7: Connectivity model of a RISC processor.

(i.e. the source operand identifiers) of the current operation have to be compared with the destination-ids of the former two operations; this requires four comparators. Essentially an *associative* search is performed of a source-id into the set of result-ids of earlier started, but not yet finished, instructions. The bypass circuit is marked by a separate box in figure 7. From this figure we determine the data path complexity, DP_{compl}, to be:

$$DP_{compl}(\text{RISC}) = (7, 2N + 4, N + 8, 3, N + 4) \tag{6}$$

and therefore the difference with the version without bypassing (the pipelined version from figure 4) is given by:

$$\Delta DP_{compl}(\text{bypassing, no-bypassing}) = (1, 0, 5, 0, 0) \tag{7}$$

and $RC_{max} = N$, and $WP_{max} = 4$, instead of 1. One bus and an extra write connection are caused by using a separate bus for immediates, which is more realistic[1]. Therefore the real difference is restricted to four write connections only. However, the increase of WP_{max} likely contributes to the critical path of the EX-stage; at the end of this stage the FU result has to go through the bypass, and may be written into one of the operand registers, which now have additional write ports.

4 Complexity of VLIW architectures

In this we apply the terminology introduced in the former section to the data path of VLIW processors, and analyze its complexity for arbitrary number of FUs (a similar analysis can be made for superscalar processors). VLIWs exploit ILP by having multiple FUs operating concurrently. The data path of a VLIW processor with two single cycle FUs is shown in figure 8. As indicated, multiple FUs may share a bus for immediate values. The figure shows one such bus; i.e. only one immediate can be specified per instruction.

[1]Usually, the source operand-ids are directly fed to the RF read buses, even if the real source operand is not a register. This enables concurrent RF access and decoding. A separate immediate bus is needed in this case.

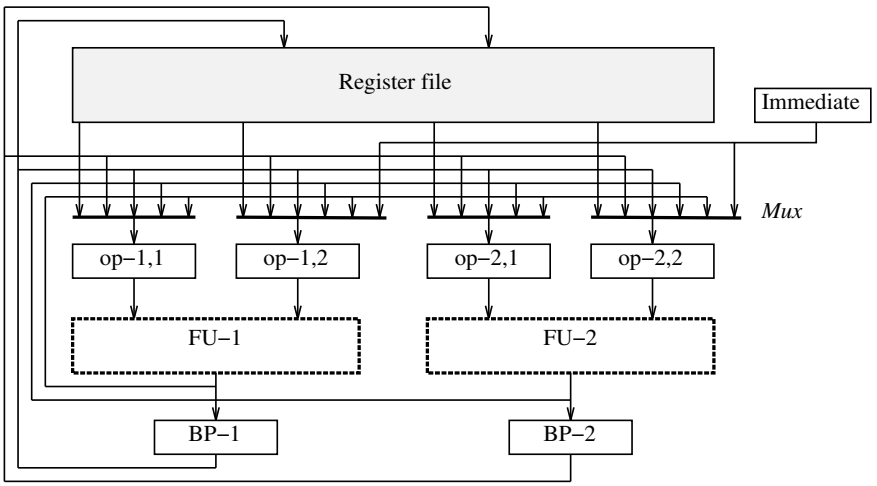


Figure 8: Data path of a VLIW processor with 2 function units.

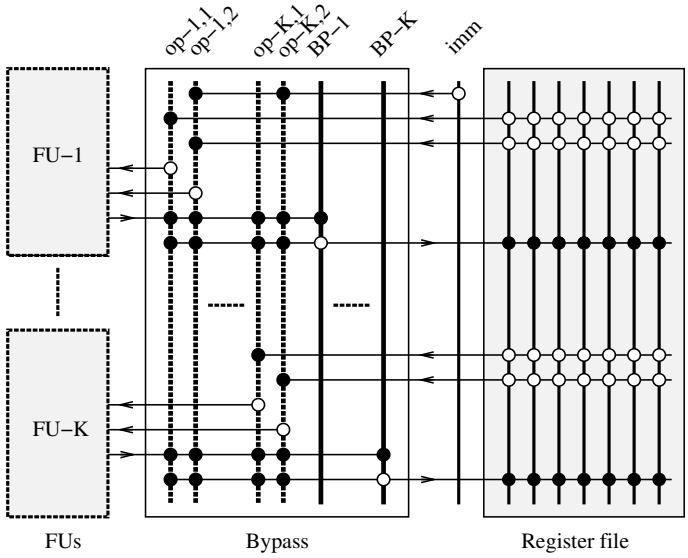


Figure 9: Connectivity model of a VLIW processor with K function units.

When more FUs are implemented multiple immediates may be required[2,3].

The connectivity model of an VLIW processor with K single cycle function units is given in figure 9. As will be clear from this figure, bypassing values from all FU outputs to their inputs requires a full crossbar connectivity. The corresponding data path complexity is given by:

$$DP_{compl}(\text{VLIW}) = (1 + 6K, 1 + 3K + 2NK, 4K + NK + 4K^2, 3K, 1 + N + 3K) \tag{8}$$

Notice the quadratic component which is typical for a crossbar. Each extra FU adds a complexity:

[2] An *orthogonal VLIW* is based on multiple RISC ALUs. This implies that each ALU is programmed by a separate 32-bit instruction field, which includes an immediate field. K ALUs require therefore K immediate buses. In figure 8 we exploit the fact that *not all ALUs require an immediate value at the same time*. This is one of the observations on which TTAs are based.

[3] The immediate may be considered to come from a separate (third) FU, which is programmed by a dedicated (immediate) field in the instruction.

$$\partial DP_{compl}(\text{VLIW})/\partial K = (6, 3 + 2N, 4 + N + 8K, 3, 3) \quad (9)$$

Furthermore, $RC_{max} = N$, $WP_{max} = 2 + 2K$, and the number of comparators needed for bypassing is given by $\#BP_{cmp} = 4K^2$. We conclude that the bypassing network complexity grows quadratically with the number of FUs. The bypassing time contains a term which is a linear function of K. Note further that adding FUs increases the RF complexity at least linearly with K.

Based on above equations we can estimate how the area and timing of bypass and RF scale with K; table 1 shows these estimates.

Unit	Metal area	Logic area	Delay
Bypass	K^2	$K^2 log(K) log(log(K))$	K
Register File	K^2	K	K

Table 1: Area and timing scaling factors; K equal the number of FUs.

In determining these scaling functions we assumed that:

1. Bus area increases linearly with the number of attached connections,
2. The area[4] of a B-bit comparator grows with $BlogB$.
3. The number of register-ids to compare with grows linearly with the number of uncompleted operations, which is of the order $O(K)$. The id-size increases therefore with $O(log(K))$.
4. The comparators are *not* in the timing critical path.
5. The critical timing increases linearly with $\#WP_{max}$. This holds as long as no hierarchical bus method is used [14]. The latter could reduce the delay to $log(\#WP_{max})$ at the cost of a much larger area.

As will be clear these scaling factors inhibit very high performance solutions.

5 Reducing complexity

As shown in the previous section, ILP processors run into problems when supporting the exploitation of large amounts of concurrency. Especially the RF and bypass components become complex when many FUs are supported. In following subsection we explore methods to reduce this complexity.

5.1 Reducing number of register ports

VLIWs with K FUs need $3K$ ports on the RF: $2K$ read ports and K write ports (assuming a VLIW with a shared RF, and FUs with two source and one destination operand). These ports are needed because in the worst case situation each FU needs to perform two reads and one write on the RF simultaneously. This amount of generated traffic on the RF can be reduced substantially however. The following cases may occur for operations, or sequences of operations.

[4]The critical timing path also increases like $BlogB$, but that fact is not used in the derivation.

1. **Fewer source operands**. Not all operations require two RF source operands. Examples are register to register copies, operations with immediate operands, loads with direct, indirect or displacement (offset) addressing, jumps, and calls.
2. **No result**. Not all operations produce a result for the RF. Examples are jumps, calls, and stores.
3. **Bypassing**. Bypassing values between FUs is needed when operations need operand values which are not yet available in the RF. It occurs when a source-id of the current operation (in the decode or register fetch stage) matches with one of the destination-ids of earlier issued, but uncompleted operations; i.e., operations for which the result(s) are not yet written back in the RF. When values are bypassed they do not have to be read from the RF.
4. **Dead result elimination**. When all usages of a value can be bypassed, it is not needed to write this value in the RF; in that case the result value is said to be *dead*.
5. **Operand sharing**. This is essentially an extension of the bypassing reasoning. An operand value may be used multiple times by succeeding operations. If the operand value is still in the bypass, the RF read traffic can be further reduced. A typical example is the case where a framepointer is used multiple times to access different operands, with different offset values, on the stack.
6. **Port sharing**. It means that the usage of a RF read port is shared by multiple reads. It occurs when multiple operations read the same register in the same cycle.

As an extreme example, that shows the difference between the actual and the worst case requirements, let us consider the following software pipelined [16] loop (omitting loop control operations) that increments and copies the elements of an integer vector:

```
    ld r1, (r2); add r2, r2, 4
    ld r1, (r2); add r2, r2, 4; add r3, r1, 1
L: ld r1, (r2); add r2, r2, 4; add r3, r1, 1; st r3, (r4); add r4, r4, 4
                               add r3, r1, 1; st r3, (r4); add r4, r4, 4
                                              st r3, (r4); add r4, r4, 4
```

The so-called *steady state* (i.e. the loop-body) of the loop is shown between two horizontal lines. Each iteration of this loop performs the operation `M(r4) := M(r1)+1`, where `M(r)` indicates the memory element at address `r`. After this operation `r1` and `r4` are incremented with four in order to point to the next vector element. A VLIW with a shared register file would need at least 5×3 RF ports. However during the *steady state*[5] of the loop (the third instruction) all operands are immediates or bypassed register operands and all write-backs are dead. Therefore the actual RF port requirement during the steady state of the loop is null!

Reducing the number of register ports assumes that we can somehow control register port usage. Two options exists for this control: dynamic (at run-time) or static (at compile-time). As shown in [1], dynamic control results in very complex logic. Since our goal is to reduce complexity we opt for static control. At compile-time it is precisely known which operands are to be accessed from the RF. We can view and implement the register file as a separate FU, called register unit (RU), having a limited number of read and

[5]The steady state is the body of a software pipelined loop.

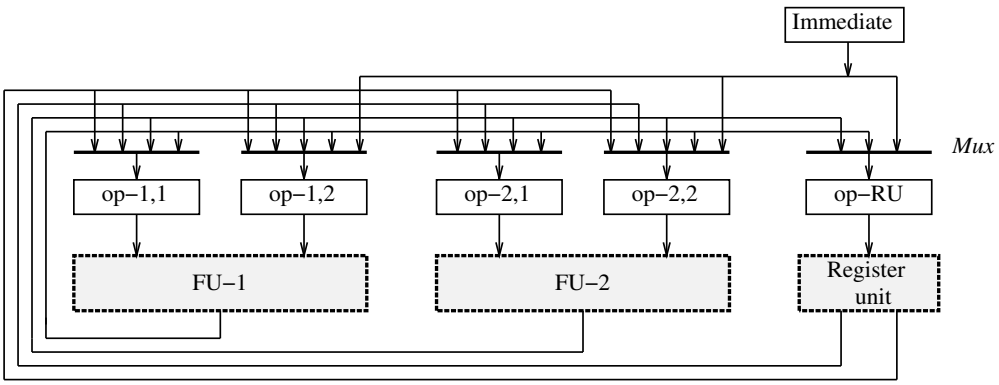


Figure 10: Data path of a VLIW with a separate RU.

write ports; e.g., figure 10 shows the data path of a VLIW with 2 FUs, and one RU having two read ports and one write port. If we compare figure 10 with figure 8 we note a remarkable difference: the bypass registers, BP-1 and BP-2, and their associated bypass buses disappeared. However, if the register read and write each take a full cycle, internal bypassing within the RU is needed, but this is much cheaper, because this bypassing is localized within one unit.

A possible instruction format for this VLIW has four fields: two fields to specify the FU operations, one for the immediate, and one field for the RU; its format is:

FU−1 field				*FU−2 field*				*imm−field*	*RU−field*		
opcode	dst	src	src	opcode	dst	src	src	immediate	dst	src	src

Bypass values are saved and restored (in the RU) under full program control; however, the bypass itself is run-time controlled.

The connectivity model of this architecture (now for K FUs, and an RU with R read and W write ports) is given in figure 11. The immediate bus is connected to one source operand register of each FU. From this figure the data path complexity is determined to be:

$$\begin{aligned} DP_{compl} &= (3K + R + W + 1, 2K + W + RN + 1, \\ &\quad 2K^2 + K(2R + W + 1) + WN) + 1, R + W, N + 2K + W + 1) \end{aligned} \tag{10}$$

Therefore, each extra FU adds the following complexity (assuming a constant number of ports):

$$\partial DP_{compl}/\partial K = (3, 2, 4K + 2R + W + 1, 0, 2) \tag{11}$$

Furthermore, the maximum number of read connections on a bus $RC_{max} = N$, and the maximum number of write ports to any register is given by: $WP_{max} = \max(W, K + R + 1)$, which reduces to $K + R + 1$ for practical situations.

If we compare these results with equations 8 and 9 for the standard VLIW, we notice a substantial reduction in connectivity complexity. Note further that the number of read and write ports are now independent design parameters which can be adapted for a specific application domain.

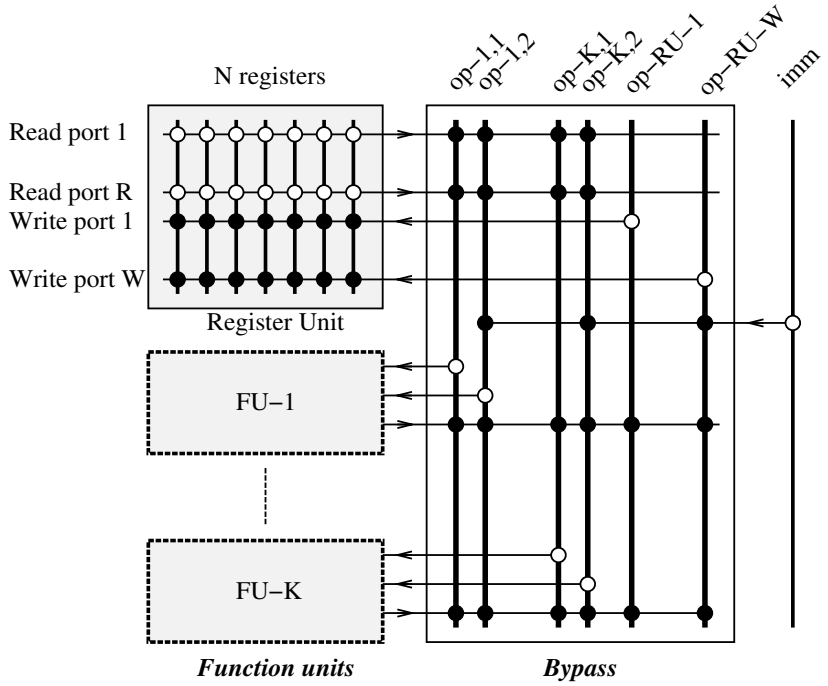


Figure 11: Connectivity model of a VLIW with separate RU.

This architecture still has a major flaw. As noted, within the bypass, all input-ids have to be matched at run-time with all output-ids. This requires a large number of comparators:

$$\#BP_{cmp} = (2K + W) \times (K + R) \tag{12}$$

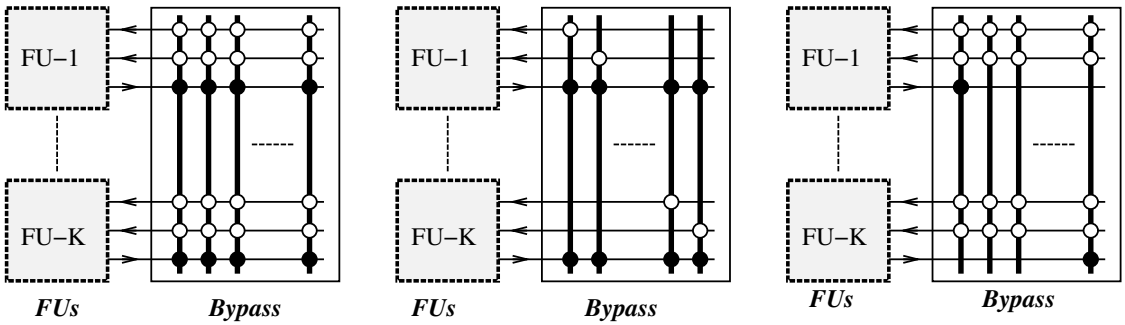
This is certainly a waste of hardware; this will be changed in the next section.

5.2 Reducing bypass complexity

Three effects contribute to the bypass complexity:

1. The associative (run-time) matching of operand identifiers.
2. The number of read and write connections; they contribute to the area and may increase the critical timing path.
3. The number of bypass buses; especially the inter FU, or *global* buses.

When treating the register file as a separate FU (i.e., as an RU), the run-time matching within the bypass is a waste of effort, because the compiler exactly knows which input the multiplexer should read from. It becomes therefore more efficient to make the bypass visible at the architectural level. Instead of a transparent register cache, the bypass becomes an explicitly addressed first-level of the memory hierarchy. Spilling of bypass values into the register unit all happens under program control. Several options exist for the resulting data path. Figure 12 shows three extreme cases: a) fully connected, b) limited read connectivity, and c) limited write connectivity. For clarity we left out the details of the RU (it is considered a separate FU), and of the immediate bus(es).



a. Fully connected b. Limited read connectivity c. Limited write connectivity

Figure 12: Making the bypass registers architecturally visible.

A fully connected bypass is not considered a real solution because the bypass is now replaced by a shared register file. This register file has to be bypassed itself as soon as we pipeline the accesses to this file; this re-introduces the shared register file problems.

Fortunately, as will be demonstrated in section 7.3, not all connections between FU inputs and outputs are equally important. For specific applications the inter FU traffic is not uniform, but heavily biased. A few transports dominate the bypass traffic. This suggests that the inter FU connectivity may be reduced, especially for embedded applications, without sacrificing performance. We will analyze the limited read connectivity case; the limited write connectivity can be analyzed similarly.

An architecture with only one read connection per bypass register is shown in figure 12b. Its bypass complexity, defined as: $BP_{compl} \equiv (\#Bus, \#RC, \#WC, RC_{max}, WP_{max}, \#Regs)$, is given by:

$$BP_{compl} = (3K, 2K, KN, 1, K, N) \tag{13}$$

where N is greater or equal to the number of operand registers ($\approx 2K$). Assuming that $N = cK$, with c some constant value, the bypass complexity increase per FU is given by:

$$\partial BP_{compl}/\partial K = (3, 2, 2cK, 0, 1, c) \tag{14}$$

Note, that 3 buses are added per FU; only one of them is non-local (for inter FU data traffic).

There are two methods to program an architecture with limited read connectivity:

1. *Explicit result method*: operations specify the bypass registers where their results should go. Source operands are implicit, except for the register file.
2. *Implicit result method*: operations put results on the bypass buses and specify explicitly from which bypass buses their source operands should be read.

The explicit result method has difficulties with specifying multicasts; i.e., when a result is needed by multiple successive operations. The implicit method does not have the multicast problem.

We started this section with noting three fundamental effects contributing to bypass complexity. Two of them, the associative matching of operand identifiers, and the number of read and write connections, have been considered; the third effect, the number of bypass buses, has not been treated yet. The number of

bypass buses still depends linearly on the number of FUs; its number can not be chosen independently. This makes it unattractive to have many FUs; e.g., splitting an ALU into its three fundamental components, an adder/subtracter, a shifter, and a logical unit, would add at least add two bypass buses. From a concurrency point of view this splitting may be attractive however. In section 6 we will see how this final problem can be handled rigorously.

6 Transport Triggered Architectures

The bypass circuit in figure 12 is still underutilized when a particular FU does not provide a result at a certain time; in other words, the bypass transport capacity is still to be designed for worst case traffic conditions. This is not a problem when at least certain code fragments require this amount of inter FU communication. However, it becomes interesting to reduce the bypass capacity (or inter FU connectivity) when the number of FU outputs is larger than the (worst case) communication requirements. This happens in the following situations:

- **FU splitting:** an FU may be split if it contains hardware for different functionality. E.g., ALUs typical contain an adder/subtracter, a shifter, and a logical unit. These units can be separated to reduce the FU resource conflicts during scheduling (when mapping operations to FUs), and allow more concurrency without a large increase of hardware. Splitting also solves the problem of having FUs which support operations with different latencies.
- **FUs with multiple outputs:** some FU operations may produce multiple results. An example is a division operation which may simultaneously produce a remainder. Usually only one result is required, however, occasionally both results may be needed.
- **Superpipelined FUs:** superpipelined FUs may contain multiple outputs for results with different latency. All these outputs can produce results in the same cycle, but that is not a typical situation.

In the latter two situations multiple outputs may share a single bypass bus; however, that means that these results can never be used in the same cycle.

To improve bypass utilization the number of buses can be reduced. Depending on the required scheduling freedom, an FU can write on a single or on multiple buses. The most significant implication of this organization is that both scheduling of operations and of transports is needed. Just as with the scheduling of register ports for the RU, two options exist: the scheduling of transports can either be done at run-time or at compile-time. Since dynamic scheduling is expensive we propose to schedule these buses at compile-time. This not only reduces complexity, but the compiler has better knowledge as to which transports have a high priority, in case there are more ready transports than available buses. Besides the bypass registers, the bypass transports become also visible at the architectural level. Figure 13 shows the connectivity model of the resulting architecture, assuming full connectivity on the bypass buses. From this model we can calculate the bypass complexity:

$$BP_{compl} = (3K + M, K(2 + M), K(1 + 2M), K, M, 2K) \tag{15}$$

where M is the number of bypass buses. Taking M constant, the incremental complexity added per FU is:

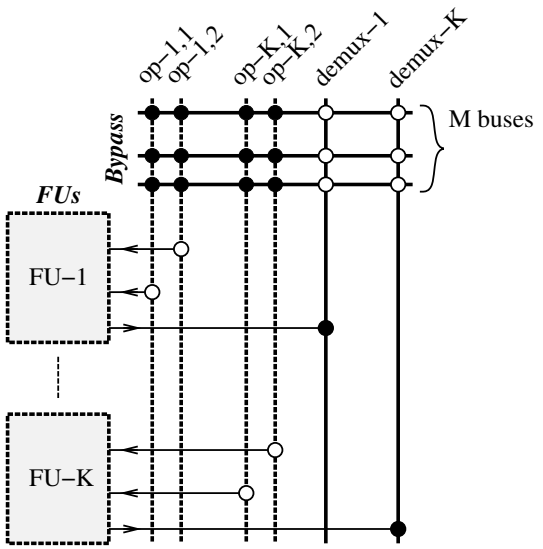


Figure 13: Connectivity model of architecture with visible bypass transports.

$$\partial BP_{compl}/\partial K = (3, 2 + M, 1 + 2M, 1, 0, 2) \tag{16}$$

We note several important implications:

1. Although the total number of buses amounts to $3K + M$, only M of them are used for global, inter FU communication. The others are local to an FU and therefore much cheaper.
2. M becomes a design parameter independent from the number of FUs.
3. The bypass complexity is, for constant M, linear with the number of FUs.
4. Given figures are for full connectivity. In practice this connectivity may be reduced. E.g., for minimal connectivity equation 15 becomes:

$$BP_{compl} = (3K, 3K, 3K, \lceil K/M \rceil, 1, 2K) \tag{17}$$

which gives an increase per FU[6]:

$$\partial BP_{compl}/\partial K = (3, 3, 3, 1/M, 0, 2) \tag{18}$$

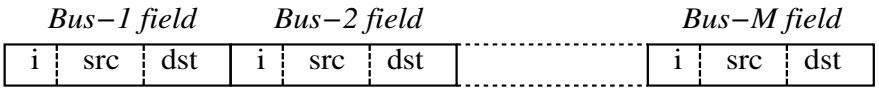
This is extremely low. Of course the necessary connectivity is highly application dependent, and will be somewhere between minimum and full connectivity. Experiments have to show what is really needed.

Making transports visible at the architectural level implies that the specification of FU operations can be hidden: data transports, also called *moves*, can trigger the FU operations implicitly; i.e., they have operations as side effect. Now we have *mirrored* the traditional programming model:

[6]The derivative of $\lceil K/M \rceil$ is of course either one or zero; its average derivative is $1/M$ as shown in equation 18.

Traditional:	Programmed operations, operation triggered transport.
New:	Programmed transport, transport triggered operations.

Traditional architectures are therefore called *operation triggered architectures* (OTAs), and the newly developed ones are named *transport triggered architectures* (TTAs). A TTA instruction may have the following format:



Each *bus-field* specifies one *move* operation from source to destination. The i-bit indicates whether the source-id has to be interpreted as an immediate or as a register specification. In general a three address operation translates into three *moves*. The next two subsections discuss several hardware and software aspects of TTAs.

6.1 Hardware aspects of TTAs

Figure 14 shows an example TTA, containing 9 function units and 6 transport buses. An FU may contain a general purpose register file; in that case it is named a register unit. Each FU is connected to the interconnection network with one or more so-called *input and output sockets*. Input sockets contain multiplexers which feed data from the buses into the FUs. Output sockets contain de-multiplexers; they put FU results on the buses. The *network controller* (NCTL) controls the pipelining of the interconnection network. A TTA is programmed by specifying the required data transports (moves). In this case each instruction contains six moves; each move specifies a transport from an FU output to one of the FU inputs (inputs and outputs may also correspond to general purpose registers).

The FU interconnection network is not limited in any sense as long as point-to-point transports can be specified. The network is not required to have full connectivity, although full connectivity would ease the code generation process. It is the task of the compiler to optimize the required transports, given a certain connectivity, such that the *cycle count* (the number of executed cycles) is minimized. Limiting the connectivity is an important tool in reducing bus loads, yielding faster transport times.

TTAs have several characteristics which make them very interesting from a hardware design point:

- *Modularity* TTAs are constructed using a restricted number of building blocks. Essentially they are built by a proper connection of FUs and bus-connections (so-called *input and output sockets*). FUs are completely independent of each other and of the interconnection network; they only have to apply to the socket interface specifications. FUs can therefore be designed separately, and pipelined independently. Each FU pipeline stage is controlled by an FU-stage controller; a chain of them implements the pipelining mechanism.

  The modularity of TTAs allows the hardware design process to be automated. Different TTAs can easily be configured by assembling different combinations of these blocks.

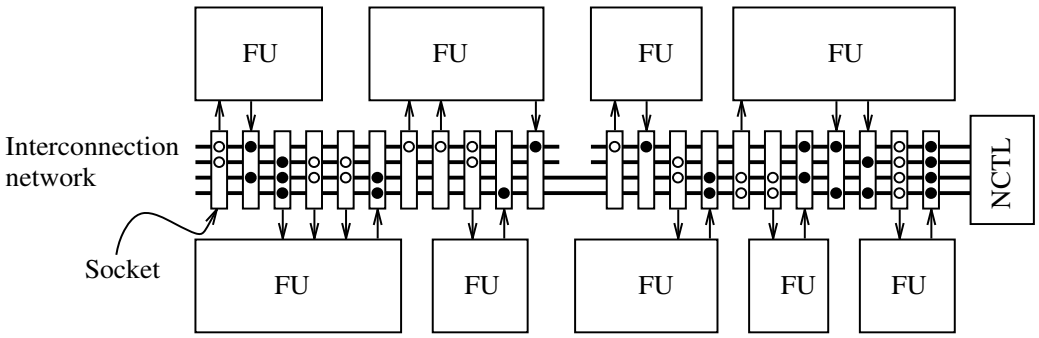


Figure 14: General structure of a TTA.

- *Flexibility and scalability* TTAs have a large flexibility; because the interconnection network is separated from the FUs, both can be designed independently; this in contrast to VLIWs where each additional FU implies a change of the interconnection network and adds extra ports to the register file. TTAs have virtually no constraints on how to design and pipeline the network, and the different FUs, as long as both obey the interface specification prescribed by the used sockets. FUs can implement any functionality, even if this requires more than two operands and multiple results.
- *Processor cycle time* If needed, the processor can be optimized for operation throughput, instead of latency. This requires superpipelining those FUs which constrain the achievable cycle time. This result is a cycle time which is limited by the time needed for inter FU data transports only. The implementation can be tuned to optimize this time by using advanced bus implementation techniques.
- *Hardware efficiency* There are several aspects contributing to an economic usage of hardware resources:

    1. *Trivial decoding*. One of the RISC design goals was to reduce decoding logic. In this respect, TTAs take the ultimate approach; they basically support only one operation format.
    2. *Transport efficiency*. Since the compiler is in control of every data transport, many transports can be optimized away.
    3. *Register efficiency*. Because many values produced during the course of a program do not need to be allocated in general purpose registers, the register file traffic rate reduces.
    4. *FU splitting*. FU logic can be split into independent parts, used for different functionality, without a large impact on the interconnection network and register file. When split, these parts may be used concurrently which results in a larger efficiency.

6.2 TTA specific compiler optimizations

Programming an operation on a TTA consists of moving operands to the input registers of an FU that is capable to perform the operation, and moving the result from the output of the FU to a RF or an FU after the operation has been performed. To illustrate TTA specific compiler optimizations, let us consider how to

translate the following code into *move code* and how to schedule it.

```
add  r1,r2,r3      // r1 = r2 + r3
sub  r4,r2,r6      // r4 = r2 - r6
st   r4,r1         // store r4 at address r1
```

First we translate each n-operand m-result operation into $n+m$ moves (n so-called *operand moves* and m *result moves*):

```
r2->O1add; r3->O2add; Radd->r1
r2->O1sub; r6->O2sub; Rsub->r4
r1->O1st ; r4->O2st
```

The notation used is: `O1add, O2add` are the operand registers of the adder, and `Radd` denotes the result (output) of this adder. Similar for the other FUs. Note that the store does not require a result move. So far, all eight moves are assumed to be executed sequentially. Scheduling consists of assigning FUs to operations and assigning cycles, sockets, and move buses to moves. Lets assume we have two FUs named `alu1` and `alu2` for ALU operations, and one FU named `ls` for load-store operations.

```
r2->O1add.alu1; r3->O2add.alu1; r2->O1sub.alu2; r6->O2sub.alu2
Radd.alu1->r1 ; Rsub.alu2->r4
r1->O1st.ls   ; r4->O2st.ls
```

The suffixes '`alu1`', '`alu2`', and '`ls`' indicate the FU on which the operation is executed. The code that we have produced so far is correct, but not optimal. A number of optimizations can be applied.

Bypassing The outputs of the add and subtract operations can be directly moved to the load-store unit. This reduces the schedule with one cycle; the number of moves does not change.

```
r2->O1add.alu1; r3->O2add.alu1; r2->O1sub.alu2    ;r6->O2sub.alu2
Radd.alu1->r1 ; Rsub.alu2->r4 ; Radd.alu1->O1st.ls;Rsub.alu2->O2st.ls
```

Dead result move elimination Next it may occur that the values in `r1` and `r2` are not life anymore because they are only used once. In that case corresponding moves can be eliminated. This gives the following schedule:

```
r2->O1add.alu1    ; r3->O2add.alu1; r2->O1sub.alu2; r6->O2sub.alu2
Radd.alu1->O1st.ls; Rsub.alu2->O2st
```

From this small example we can make an important observation. We did not need 3 moves and 3 RF accesses for each operation. In this example we totally needed 6 moves and 4 RF accesses for three operations instead of 9 transports and 9 RF accesses.

Operand sharing This occurs when succeeding operations on the same FU use the same operand. For example, in the following code fragment the second operand move is redundant and can be eliminated since the value of `r1` is already in `O1add.alu`.

```
r1->O1add.alu; r2->O2add.alu
r1->O1add.alu; r3->O2add.alu; Radd.alu -> r2
Radd.alu -> r3
```

An interesting case of operand sharing may occur within loops, where the shared operand is the offset value used for the address or counter increment operation; the transport of this offset to the FU may turn out to be loop-invariant; in that case it can be put in the loop-prologue instead of the loop-body.

Socket sharing Sometimes the usage of an output socket can be shared by multiple reads; this is called socket sharing. It occurs when reading the same register on different buses, or when accessing different registers which are connected to the same socket, in case these accesses are mutually exclusive. In both cases the number of moves is not reduced, but a schedule may be generated which was otherwise inhibited by hardware resource constraints. As demonstrated by experiment 2, socket sharing substantially reduces the number of register ports.

Scheduling freedom A final optimization concerns the extra scheduling freedom when scheduling moves instead of operations. E.g. source operands do not have to be moved in the same cycle. Also, depending on the implementation of the FUs, there can be more or less freedom in scheduling the transports of the FU result values. This scheduling freedom has two effects: 1) it enhances the application of described optimizations, and 2) values can stay longer in the FUs; this lowers the demands on the register file.

7 Experimental evaluation

In the former sections it was argued that making data transports visible at the architectural level results in a number of advantages. Three of the main advantages will be measured in this section: reduction of the number of required register ports, the extra concurrency available when splitting FUs, and the reduction of bypass connectivity. For these measurements we use a highly ILP optimizing TTA compiler and an architecture exploration tool, both developed by Hoogerbrugge [17]. The first two measurements, presented in sections 7.1 and 7.2, use the following set of benchmarks:

> *A68, Bison, Cjpeg, Cmp, Compress, Cpp, Dhrystone, Diff, Djpeg, Expand, Flex, Gawk, Grep Gs, Gzip, Linpack, Livermore, Mpeg_play, Od, Queen, Sed, Sieve, Sort, Stanford, Sum, Tr Unexpand, Uniq, Virtex*, and, *Wc*.

These benchmarks are well known within the 'UNIX world'. They mainly contain integer type of code, except for Linpack and Livermore, which contain substantial amounts of floating point code as well. For the measurements in section 7.3 we will focus on a high performance image processing application.

7.1 Register file traffic reduction

One of the main claims we have made about TTAs is that they reduce the number of register ports. This section presents measurements which support this claim. The architecture that we shall use is rather large: it contains twelve move buses, eight FUs, and four long immediates; each FU supports all operations. The number of read and write ports is varied between one and six. The motivation for this architecture is that we do not want to limit the performance by other resources than register ports.

In the experiment we vary the number of read and write ports between 1 and 6, giving 36 different RU configurations. Figure 15 shows how many operations are executed per cycle for each of the 36 configurations. This experiment shows extremely good results. For example, even with the one read and one write port RU two operations per cycle can be executed. A VLIW or superscalar requires at least three FUs to reach this performance; it needs therefore minimally six read and three write ports on its RU(s). A TTA saves in this case 78% on the number of RU ports.

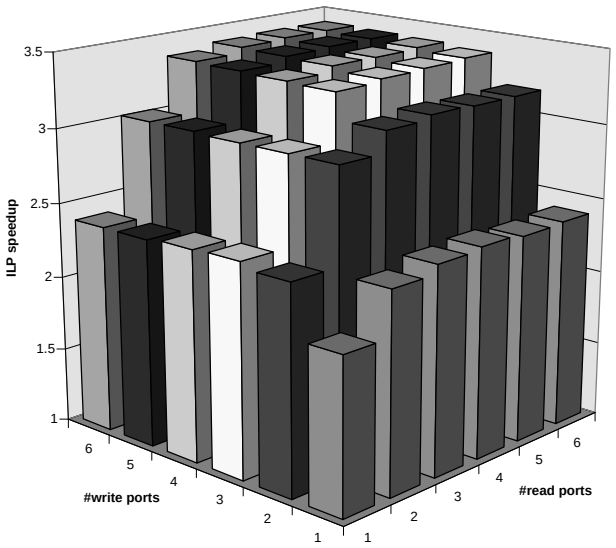


Figure 15: Operations per cycle as function of the number of RU ports.

This experiment also tells us the optimal division between read and write ports for a certain number of ports, for the given set of benchmarks. For example the optimal configuration for a 5 ported RU consists of 3 write ports and 2 read ports. It is remarkable that some of the optimal configurations have more write ports than read ports; we would have expected the opposite. This is due to dead result move elimination. When a result move is scheduled a write port is claimed. However, after scheduling the operations that are flow dependent on it, the result move can become dead and the claimed write port is released. Therefore the scheduler sometimes claims an extra write port during scheduling that is not needed in the final code. A different scheduling approach, which avoids this, will be used in future compilers [18].

7.2 FU splitting

One of the advantages of TTAs is that different functionality (like arithmetic and logical operations) does not have to be combined into one FU (e.g. an ALU); it is easier to have separate FUs for non-related operations.

This results both into design and performance advantages. The design becomes more modular; e.g., different numbers of adders and comparators can be used in a design. The performance is enhanced because multiple operations can be started in a single cycle, resulting in a better FU utilization.

To measure the effect of FU splitting we scheduled the benchmarks for two TTAs: a *small* one and a *medium* sized TTA. For each TTA two versions are used: an *unsplit* and a *spit* one. The small, unsplit TTA contains one (single cycle) ALU executing all integer type of operations including all types of load and stores, and one FPU (floating point unit), which supports all floating point operations. The small, split TTA has the same functionality except that its FUs are split; the ALU is split into five separate FUs for arithmetic, logic, shift, compare, and one for load and store operations. Similar, the FPU is split into two FUs; one for multiply and divide, and the other supporting all remaining floating point operations (except the load and stores of floating point numbers which are handled by the integer units). The medium sized, unsplit and split TTAs are similar except that their integer functionality is doubled (2 ALUs, etc.). During the measurements the number of buses is varied from one to ten.

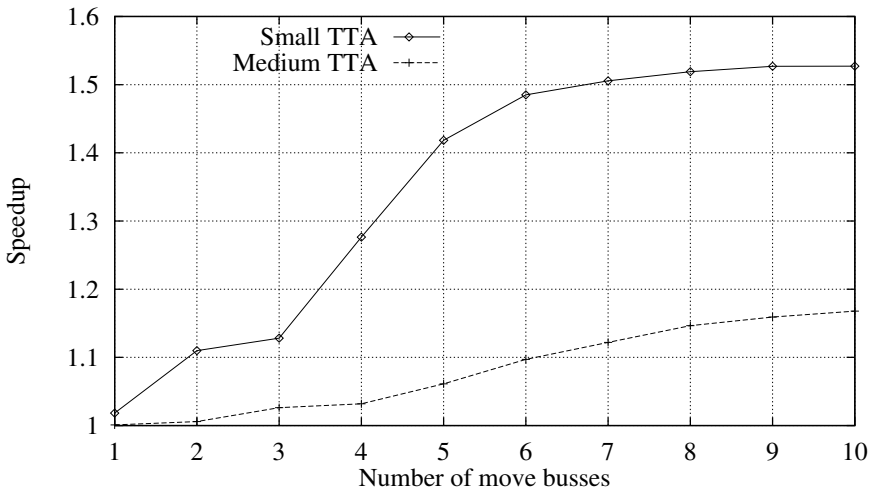


Figure 16: Speedups achieved when splitting FUs.

Figure 16 plots the speedup achieved when splitting FU functionality. Clearly for a small number of buses the performance gain is not large, because in that case even the unsplit TTA contains sufficient FUs to utilize all available transport capacity. However, for a larger number of buses the speedup can easily be more than 15% for the medium sized TTA, and more than 50% for the small one. For more parallel applications the gains will certainly be larger. Since the costs of FU splitting are marginal, it requires only extra sockets[7], we may conclude that indeed FU splitting is a key TTA feature which should certainly be exploited.

7.3 Bypass connectivity reduction

In this section we demonstrate the extreme power of TTAs in reducing bypass connectivity, i.e., the connectivity between FUs and Register files. As an example we take a set of high performance image processing ap-

[7]Sockets act as (de-)multiplexers. In fact, when FUs are not split we need also (de-)multiplexers in order being able to distribute the operands to the different logic units (contained within one FU) and to extract the results. So the hardware advantage of unsplit TTAs is really marginal.

plications: *color conversion* (from one color spase to another using tri-linear interpolation), *edge-detection*, and *convolution* (like required for sharpening and smoothing) [19].

An architecture with optimal cost-performance ratio has been developed for these set of applications using the architecture exploration tool from the MOVE framework (as described in [17, 20]). This resulted in a TTA with 13 arithmetic FUs, as shown in figure 17. Some FUs were special designed: a four-input 'Multimul' unit performing the operation $e = (a \times b) + (c \times d)$; min-max units, units for accessing local line buffers, for table-lookup, for byte-packing, and units for pixel-stream input and output. Originally, all 14 buses are connected to all FU and register file inputs and outputs, requiring 938 bypass connections. The shown architecture requires 5 cycles/pixel for color conversion, 3 cycles/pixel for convolution and 7 cycles/pixel for edge detection.

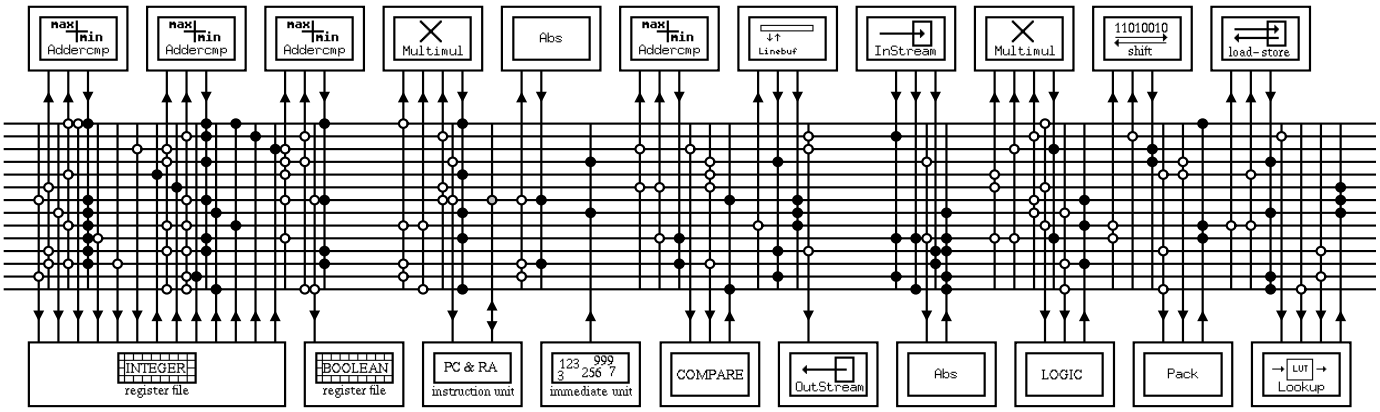


Figure 17: An optimized TTA for different image processing applications.

The results of the connectivity reduction design step are plotted in figure 18. The figure shows the execution time as a function of the number of removed connections. The execution time first drops as a result of reducing bus load; the total capacitance on the buses reduces and therefore the achievable cycle time. The optimal architecture is found when 771 connections are removed, i.e. 82% of all the connections. Removing more connections increases the number of executed cycles because the compiler can not any more schedule optimal code. Note, that the more specialized the application, the more connections can be removed.

One problem remains; looking at figure17 we still observe many ports and a high connectivity on the (integer) register file. However, as we have shown in [21], this problem can be solved by splitting the multi-ported register file in multiple single ported files. Our compiler can still schedule for such an architecture with hardly any performance penalty.

8 Summary and Conclusions

This paper analyzed the data path complexity problems of several ILP architectures. It introduced several complexity measures to perform this analysis. It showed that the data path of ILP architectures becomes very complex when scaled to high performance levels.

Methods were researched to reduce the complexity of the register file and the bypass circuitry. Their

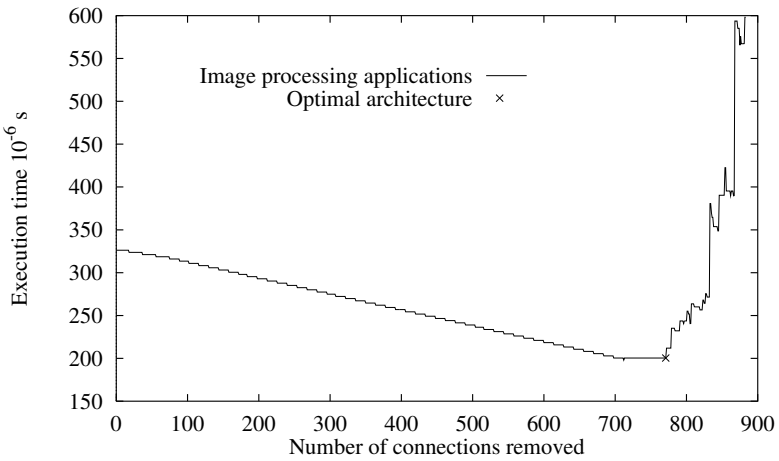


Figure 18: Connectivity reduction design step.

complexity could be substantially reduced by adding a new level of control, the data path control level, to the compiler, and removing this control from the hardware. This resulted in a new class of architectures, called transport triggered architectures (TTAs), which has been shown to have many advantages, both at the hardware and the software level.

It has been shown that one important bottleneck of VLIWs, the large number of required ports on the register file, can be substantially reduced. Furthermore, 15-50% speedups have been demonstrated by splitting FUs into multiple independent components. For traditional VLIWs and superscalars this splitting is difficult, because it directly results into more complexity of the register file and the FU interconnection network. Finally, it has been demonstrated that the connectivity of the bypass network can be reduced with more than 80% without any performance loss. In fact the performance increased due to shorter achievable cycle times.

TTAs have a high flexibility and scalability; they cover a very large design space. These characteristics make them especially suitable in the application specific computation domain (for embedded systems). Their modularity make them a good target for *automatic generation* as well, giving a very short design trajectory. A framework has been designed for the automatic mapping of arbitrary C/C++ applications into a TTA hardware and corresponding object code [17, 20]. A number of TTAs have currently been designed [22, 19]. Their realization confirmed the explained advantages.

References

[1] William M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[2] Arvind and D.E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[3] David E. Culler et al. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of ASPLOS-IV*, pages 164–175, Santa Clara, California, April 1991.

[4] G. M. Papadopoulos and K. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *18th Int. Symp. on Computer Architecture*, Toronto, May 1991.

[5] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *ISCA'20 proceedings*, pages 224–235, May 1993.

[6] Anant Agarwal et al. Sparcle: an evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, pages 48–61, June 1993.

[7] Gurinda S. Sohi, Scott E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *ISCA'22*, pages 414–425, Santa Margherita Ligure, Italy, June 1995.

[8] Pradeep K. Dubey, Kevin O'Brien, Kathryn O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 109–121, 1995.

[9] Jenn-Yuan Tsai and Pen-Chung Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking adn control speculation. Technical Report TR 96-037, Univ. of Minnesota, Department of Computer Science, 1996.

[10] Jeroen Hordijk and Henk Corporaal. The Impact of Data Communication and Control Synchronization on Coarse-Grain Task Parallelism. In *Second Annual Conf. of ASCI*, Lommel, Belgium, June 1996.

[11] Z. Li, J.Y. Tsai, X. Wang, P.C. Yew, and B. Zheng. Compiler techniques for concurrent multithreading with hardware speculation support. In *Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing*, August 1996.

[12] M. R. Thistle and B. J. Smith. A processor architecture for horizon. *Proc. Supercomputing*, pages 35–41, November 1988.

[13] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI*. Addison Wesley, 1990.

[14] Henk Corporaal. *Transport Triggered Architectures; Design and Evaluation*. PhD thesis, Delft Univ. of Technology, September 1995. ISBN 90-9008662-5.

[15] John L. Hennessy and David A. Patterson. *COMPUTER ARCHITECTURE, a Quantitative Approach, Second Edition*. Morgan Kaufmann publishers, 1995.

[16] Monica Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[17] Jan Hoogerbrugge. *Code generation for Transport Triggered Architectures*. PhD thesis, Delft Univ. of Technology, February 1996. ISBN 90-9009002-9.

[18] Johan Janssen and Henk Corporaal. Registers on demand: Integrated register allocation and instruction scheduling. In *Third Annual Conference of ASCI*, The Netherlands, June 1997.

[19] Marnix Arnold, Reinoud Lamberts, and Henk Corporaal. High Performance Image Processing using TTAs. In *Second Annual Conf. of ASCI*, Lommel, Belgium, June 1996.

[20] Henk Corporaal and Jan Hoogerbrugge. Cosynthesis with the MOVE framework. In *CESA'96 IMACS Multiconference*, pages 184–189, Lille, France, 1996.

[21] Johan Janssen and Henk Corporaal. Partitioned Register Files for TTAs. In *Proceedings of the 28th Annual Workshop on Microprogramming*, Michigan, November 1995.

[22] Herman Adriani, Françoise Harmsze, and Henk Corporaal. The utilization of a fully configurable microprocessor development environment for rapid VHDL prototyping and implementation of 'C'-based algorithms. In *Proceedings of ProRisc workshop*, Mierlo, The Netherlands, November 1996.