ILP architectures: trading hardware for software complexity

Henk Corporaal
Delft University of Technology, Electrical Engineering Dep.
Mekelweg 4, 2628 CD Delft, The Netherlands
E-mail: heco@cs.et.tudelft.nl

Currently, several interesting superscalar and VLIW (very large instruction word) processors hit the market. These processors exploit so-called instruction level parallelism (ILP); each cycle multiple operations are executed. This paper analyzes the data path complexity of ILP processors; in particular of VLIWs. It demonstrates that their complexity gets out of control when scaling to very high performance. Several methods are researched for reducing this complexity. Essentially these methods trade hardware for software complexity, i.e., performing as much as possible at compile time. This results in a new architectural approach called transport triggering. Its concept and characteristics are outlined. The application of this concept results in a number of hardware advantages, and introduces several new scheduling optimizations.

1 Introduction

The history of the last decade showed two important developments with respect to the design of more powerful computers: the change from CISC to RISC, and the exploitation of ILP (instruction level parallelism) in Superscalar and VLIW processors. RISC design principles aimed at reducing instruction set complexity such that instructions can easily be pipelined and throughput maximized. Simple and easy to design processors were the result. However, with the exploitation of ILP, processors again become increasingly complex. Superscalar processors issue multiple instructions per cycle; this requires complex logic which detects and resolves (using register renaming) dependences between these instructions, and for hardware resource allocation.

VLIWs take a different approach; here the compiler is responsible for allocation and depence checking. It puts multiple operations into each instruction. Only one instruction is issued per cycle. VLIWs have better scaling properties. In addition they are more flexible; their functionality can easily be adapted to specific applications. Therefore they are good candidates for high performance low cost embedded systems. Several interesting and very powerful VLIWs hit the (multi-media) market: the Trimedia of Philips, the Mpact of Chromatic and the TMS320C6x of Texas Instruments. Despite their good properties, the data path of VLIWs may become 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: *transport triggering*. The paper is structured as follows. Section 2 introduces the required terminology concerning the complexity of the data path of ILP processors and applies this terminology to RISC processors. Section 3 treats the complexity of VLIW processors. Section 4 demonstrates several methods to reduce ILP processor data path complexity. Section 5 describes an approach we have taken to reduce complexity: transport triggered architectures are described. Finally, section 6 draws major conclusions.

2 Data path complexity: terminology

In general an instruction specifies one or more operations, and for each operation zero or more source and 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 in a simple, 4-stage pipelined, RISC type of processor. Its pipelining diagram is shown in figure 1. Its stages are: instruction fetch (IF), instruction decode and register fetch (DC), execution (EX), and write-back of results into the general purpose register file (WB).

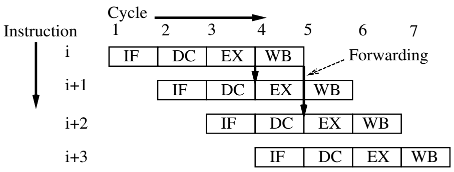


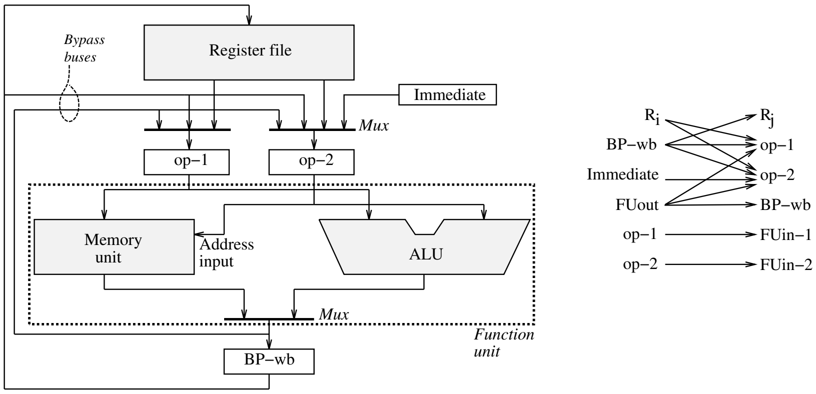
Figure 1: RISC pipelining diagram (4-stage pipeline).

A fundamental performance bottleneck results when pipelining processors: results of function units (FUs) have to be forwarded in order to get single cycle instruction throughput[1]. Forwarding of results at the end of the EX and WB-stage is indicated in figure 1. Forwarding support requires a so-called *bypass circuit*.

A possible data path of this processor is given in figure 2a. Its 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.

To enable all required data transports a certain connectivity between source and destination locations of operands is required. Figure 2b shows the connectivity graph (CG) for this RISC processor. Although this graph shows which sources and destinations need direct connections, it does not tell how to implement this connectivity. In general many options for the data path exist, ranging from a single bus connecting all sources and destinations, to separate connections for each edge in CG. In order to determine its complexity we define the data path complexity.

Definition 2.1 *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*



a. Data path b. Connectivity graph

Figure 2: RISC data path and connectivity graph.

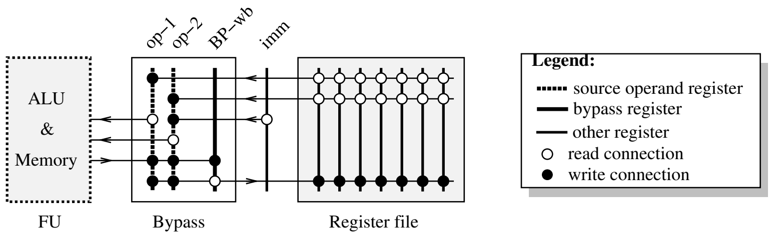


Figure 3: Connectivity model of a RISC processor.

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[2,3]. We therefore also define:

Definition 2.2 *RC_{max}, the maximum number of read connections to any bus, and WP_{max}, the maximum number of write ports to any register.*

The data path complexity of the discussed RISC processor can easily be determined from figure 3; this figure shows a different view of the data path, including all the necessary read and write connections; we call it the *connectivity model* of the processor. 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 (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. From this figure we determine the data path complexity, DP_{compl}, to be:

$$DP_{compl}(\text{RISC}) = (7, 2N+4, N+8, 3, N+4) \tag{1}$$

where N equal the number of general purpose registers; furthermore, $RC_{max} = N$ and $WP_{max} = 4$.

A few remarks are appropriate at this point: (1) 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; (2) of the seven buses four are very simple; they only serve to connect FU in and outputs to corresponding source operand and result registers, and for the transport of immediate values; and (3) 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.

3 Complexity of VLIW architectures

In this section we research the complexity of VLIW architectures, and apply the terminology introduced in the former section. We analyze its complexity for arbitrary number of FUs (the analysis of a superscalar data path can be made similarly).

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 4. 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. When more FUs are implemented multiple immediates may be required.

The connectivity model of an VLIW processor with K single cycle function units is given in figure 5. 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:

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

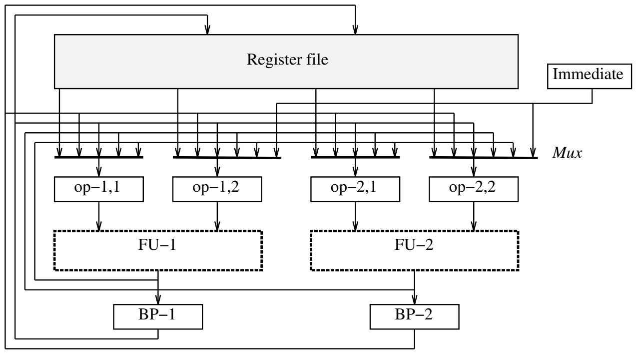


Figure 4: Data path of a VLIW processor with 2 function units.

We observe that the bypass circuit 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. Although a VLIW has a high performance potential, its bypassing network complexity (both in area and time) gets out of control for larger values of K. Also the shared RF turns out to be a fundamental bottleneck. Based on above equations we can estimate how the area and timing of bypass and RF scale with K; table 1 shows these estimates.

In determining these scaling functions we assumed that: (1) bus area increases linearly with the number of attached connections; (2) the area 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 linear in K. The id-size increases therefore with $O(logK)$; (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 [3]. 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.

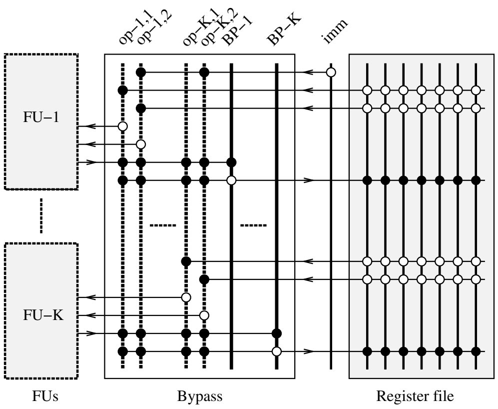


Figure 5: Connectivity model of a VLIW processor with K function units.

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.

4 Reducing complexity

In this section we explore methods to reduce the register file (RF) and bypass complexity.

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.

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.

No result. Not all operations produce a result for the RF. Examples are jumps, calls, and stores.

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.

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*.

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.

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.

The factors contributing to the difference between actual and worst case requirements for the number of register ports on the RF have been measured for a set of 15 scalar type (integer code) benchmarks (*Bison, Cjpeg, Cpp, Dhrystone, Diff, Djpeg, Expand,*

Case	Measurement	Result
1	Average number of RF source operands for sequential code	0.97
2	Average number of RF results for sequential code	0.60
3	Percentage of source operands that are bypassed	50%
4	Percentage of results that are eliminated by dead result elimination	35%
5	Percentage of RF operands that can be shared	8.9%
6	Average number of RF reads that share the same port	1.23
Overall results		
Reads	Average number of RF source operands for parallel code	0.50
Writes	Average number of RF results for parallel code	0.35
Read Ports	Average number of RF read ports for parallel code	0.41

Table 2: Factors contributing to register file traffic reduction.

Od, Sed, Sort, Stanford, Tr, Unexpand, Uniq, and *Virtex*). The architecture, for which the benchmarks are compiled, is assumed to have sufficient hardware resources.

Table 2 shows the contribution to RF traffic reduction for each of the 6 listed cases. The first two lines make clear that a lot of operations do not need two RF operands, or do not produce a result. The average number is 0.97 RF operands and 0.60 results per operation. The following four lines show that the contribution of bypassing, dead result elimination, and port sharing is substantial. The contribution of operand sharing is low since it does not occur very frequently in scalar type code, and about half of the common operands are immediate values. After the scheduler schedules the code and applies bypassing, dead result elimination, operand sharing, and port sharing the number of RF reads and writes per operation become 0.50 and 0.35. If we compare this with the original, worst case, values of 2 reads and 1 write per operation, we see that read traffic is reduced by 75% and write traffic by 65%. As a result of port sharing the average number of RF read ports per operation reduces to 0.41.

From this experiment we conclude that the average register requirements are far lower than worst case. Therefore we may reduce the number of register ports (and also somewhat the number of required registers), assuming we can control register port usage. Two options exists for this control: dynamic (at run-time) or static (at compile-time). As shown by Johnson[4], 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 write ports. 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 such an architecture (with K FUs, and an RU with R read and W write ports) is given in figure 6. The immediate bus is connected to one source operand register of each FU. From this figure its data path complexity is easily determined:

$$DP_{compl} = (3K + R + W + 1, 2K + W + RN + 1, \\ 2K^2 + K(2R + W + 1) + WN) + 1, R + W, N + 2K + W + 1) \quad (3)$$

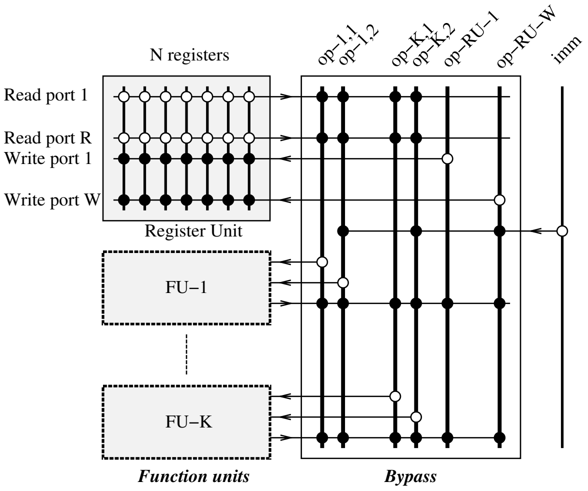


Figure 6: Connectivity model of a VLIW with separate RU.

Therefore, each extra FU adds the following complexity (assuming a constant number of ports). 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 with equation 2 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.

This architecture still has a major flaw, though. 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, namely $(2K + W) \times (K + R)$. This is certainly a waste of hardware; this will be corrected in the next section.

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, and (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

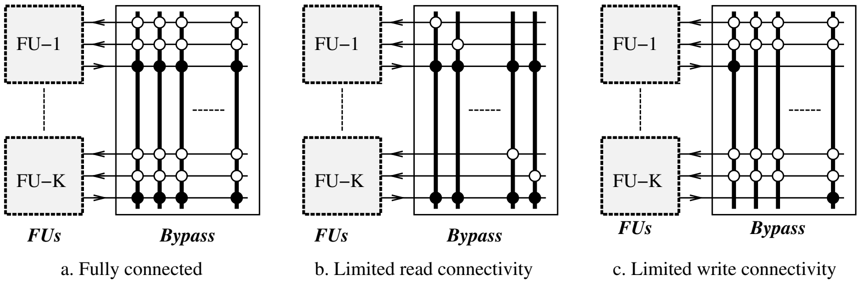


Figure 7: Making the bypass registers architecturally visible.

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 7 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 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, not all connections between FU inputs and outputs are equally important; usually only a few connections are responsible for most of the traffic; this holds even stronger for embedded system applications. This suggests that the inter FU connectivity may be reduced 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 7b. Its bypass complexity, which we define as: $BP_{compl} \equiv (\#Bus, \#RC, \#WC, RC_{max}, WP_{max}, \#Regs)$, is given by:

$$BP_{compl} = (3K, 2K, KN, 1, K, N) \tag{4}$$

where N is greater or equal to the number of operand registers ($\approx 2K$). Note, that 3 buses are added per FU; only one of them is non-local (for inter FU data traffic).

5 Transport Triggered Architectures

Although we removed the associative matching of operand identifiers and were able to reduce the number of bypass connections, the bypass circuit in figure 7 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.

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.

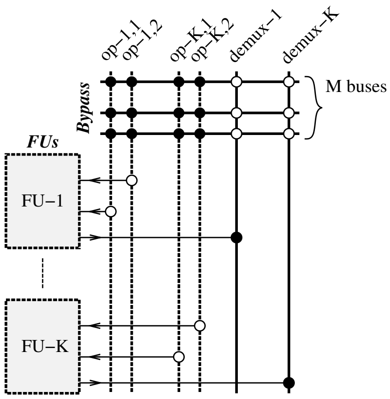


Figure 8: Connectivity model of architecture with visible bypass transports.

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 scheduling 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 8 shows the connectivity model of the resulting architecture, assuming full connectivity on the bypass buses. From this model we can calculate the bypass complexity:

$$BP_{compl} = (3K + M, K(2 + M), K(1 + 2M), K, M, 2K) \tag{5}$$

where M is the number of bypass buses.

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; and (4), given figures are for full connectivity. In practice this connectivity may be reduced. E.g., for minimal connectivity equation 5 becomes:

$$BP_{compl} = (3K, 3K, 3K, \lceil K/M \rceil, 1, 2K) \tag{6}$$

This is extremely low. Of course the necessary connectivity is highly application dependent, and will be somewhere between minimum and full connectivity.

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: traditionally we program operations which trigger transports; now we program transports which trigger (FU) 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:

Bus−1 field			*Bus−2 field*				*Bus−M field*		
i	src	dst	i	src	dst	......	i	src	dst

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*.

Hardware aspects of TTAs

Figure 9 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*. 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. It is the task of the compiler to optimize the required transports, given a certain network 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 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. The modularity of TTAs allows the hardware design process to be automated.

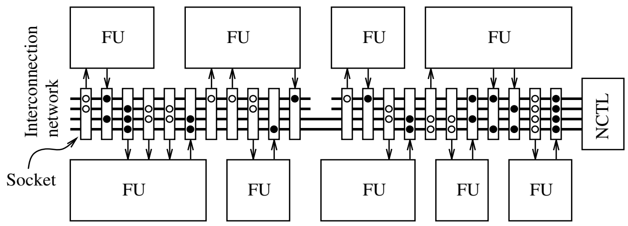


Figure 9: General structure of a TTA.

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.

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.

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; and (4) *FU splitting*. FU logic can be split into independent parts, used concurrently, without a large impact on the interconnection network and register file.

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 schedule the following code:

```
add  r1,r2,r3   // r1=r2+r3 //  r2->O1add; r3->O2add; Radd->r1
sub  r4,r2,r6   // r4=r2-r6 //  r2->O1sub; r6->O2sub; Rsub->r4
st   r4,r1      // store r4 //  r1->O1st ; r4->O2st
```

The RISC code (left) is translated into TTA code (right): each n-operand m-result operation into $n + m$ moves (n so-called *operand moves* and m *result moves*). 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 total we 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.

Port sharing: Sometimes the usage of an output socket can be shared by multiple reads; 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. The number of moves is not reduced, but a schedule may be generated which was otherwise inhibited by hardware resource constraints. As shown in table 2, port sharing substantially reduces the number of required 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 enhances the application of described optimizations. Values can also stay longer in the FUs; this lowers the demands on the register file.

6 Conclusions

This paper analyzed the complexity problems of several ILP architectures. It introduced 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 circuit. Their 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. A number of TTAs have currently been designed[5,6]. Their realization confirmed the explained hardware advantages.

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[7,8].

1. John L. Hennessy and David A. Patterson. *Computer Architecture, a Quantitative Approach, Second Edition*. Morgan Kaufmann publishers, 1995.
2. H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI*. Addison-Wesley, 1990.
3. Henk Corporaal. *Transport Triggered Architectures; Design and Evaluation*. PhD thesis, Delft Univ. of Technology, September 1995. ISBN 90-9008662-5.
4. William M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.
5. 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.
6. Marnix Arnold, Reinoud Lamberts, and Henk Corporaal. High Performance Image Processing using TTAs. In *Second Annual Conference of ASCI*, pages 68–75, Lommel, Belgium, June 1996.
7. Jan Hoogerbrugge. *Code generation for Transport Triggered Architectures*. PhD thesis, Delft Univ. of Technology, February 1996. ISBN 90-9009002-9.
8. Henk Corporaal and Jan Hoogerbrugge. Cosynthesis with the MOVE framework. In *CESA'96 IMACS Multiconference*, pages 184–189, Lille, France, 1996.