Structured Hardware Compilation of Parallel Programs

Wayne Luk, David Ferguson and Ian Page

Programming Research Group, Oxford University Computing Laboratory,
11 Keble Road, Oxford, England OX1 3QD

Abstract

A major bottleneck in automatic hardware synthesis is the time to place and route the netlist produced by a hardware compiler. This paper presents a method which exploits the syntax of the source program to guide its layout in a device-independent manner. The technique has been used in prototyping a hardware compiler for a commercially-available device, the Algotronix CAL1024 Field-Programmable Gate Array. The potential of this approach is evaluated.

INTRODUCTION

Recent work has shown how parallel programs, in a language such as occam (Inmos 1984), can be compiled into synchronous and asynchronous circuits (Page and Luk 1991, Weber *et al* 1992). This approach enables the rapid generation of custom hardware from a high-level description; it also provides a uniform framework for developing and verifying systems with both hardware and software components.

At present many hardware compilers produce from a source program a netlist description which specifies the appropriate connection of circuits in the target technology. The subsequent placement and routing of the components in the netlist is often very time-consuming and does not always result in a desirable circuit. There is no simple way of predicting reasonably accurately the size of the implementation without going through detailed placement and routing. We are exploring various methods which may overcome these drawbacks; one possibility will be described in this paper.

The essence of our experimental compilation scheme is to exploit the structure of the source program to guide the layout of the target implementation. A circuit module is developed for each control structure, such as sequential or parallel composition, of a subset of occam; a layout convention is then adopted to connect these control modules to registers and to arithmetic circuits using parametrised wiring blocks. The compilation scheme can be described succinctly as a mapping from the source language to the OAL language (Luk and Page 1991), which specifies the relative placement of components. Techniques for estimating the size and performance of the compiled circuit are also available.

While the proposed compilation scheme is device-independent, to demonstrate its viability a prototype compiler has been written for Algotronix's CAL1024 FPGAs (Algotronix 1990). Some examples, such as run-length coding, have been developed using this compiler. Currently no optimisations have been implemented in our prototype compiler, and the layouts produced are often rather large; methods to reduce their size will be discussed.

APPROACH

The source language for our compilation scheme is a variant of the occam language. Occam has been chosen because of its simplicity, its expressive power for parallelism, and its well-defined semantics (Page and Luk 1991). Control structures in our source language include variable assignment and communication, sequential and parallel composition, and the conditional, iteration and prioritised alternation statements. To simplify the construction of the compiler and to produce fast implementations, fully parallel hardware is generated for expression evaluation, and a distributed control scheme is adopted to ensure the correct sequencing of commands.

As an example, the control circuit for SEQ P Q, the sequential composition of P and Q, must ensure that the hardware for Q is activated only after activities in hardware for P have terminated. This can be achieved through a system of signals called *request* and *acknowledgement* (Weber *et al* 1992), or *start* and *finish* (Page and Luk 1991). The synchronous version of this control scheme – which is what we shall focus on in this paper – specifies that to activate a block, its request line req is raised for one clock cycle; when the block has completed its activities, it raises its acknowledgement line ack for one cycle to activate the next segment of the control circuitry. Hence to implement SEQ P Q, the ack of P is directly connected to the req of Q (Figure 1(a), where req and ack are abbreviated to r and a).

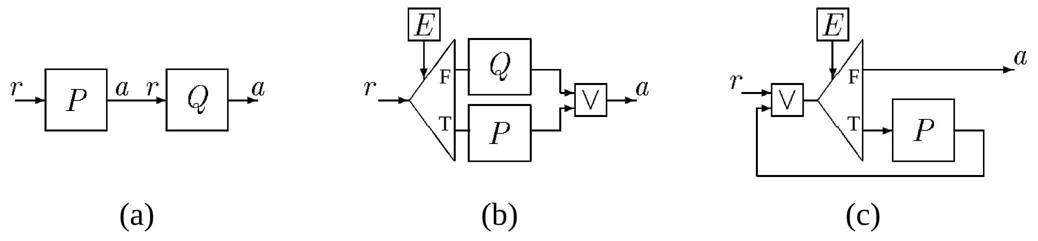


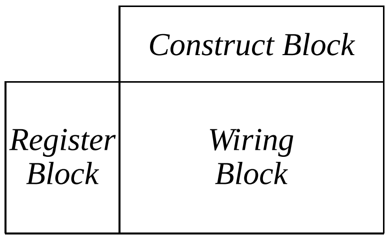
Figure 1 Hardware for (a) SEQ P Q, (b) IF E P TRUE Q (if E then P else Q) and (c) WHILE E P (while E do P). The $\vee$-labelled component is an or-gate. The triangular-shaped component is a demultiplexer, which steers its horizontal input to either the T (true) or F (false) output depending on the boolean expression E.

Other statements can be compiled in a similar manner; the control circuitry for PAR P Q, for instance, consists of a fulladder whose inputs are the ack signals from P and Q and its own latched sum output, so that its carry output will produce an acknowledgement signal when both P and Q have acknowledged. Two more examples are given in Figure 1, which shows the control hardware for the IF and WHILE constructs (to avoid combinational loops, the ack output for WHILE may be latched). One can regard our control scheme as a token-passing system, whose activity at a particular instant is governed by the component in possession of a token. The initial token is produced by a block called *starter* and there can be more than one token in circulation for programs containing one or more PAR commands.

Next, we need to lay out the circuit blocks generated from the source program. The traditional method is to use automatic placement and routing software supplied by the device manufacturer; the result is usually of acceptable quality, but can take a long time to produce. A different approach has been adopted in this project: a fixed floorplan format is used to lay out the components in the netlist. This allows the rapid generation of placed and routed circuits, possibly at the expense of compactness and performance. The following describes our techniques in greater detail.

COMPILATION STRATEGY

To simplify the presentation, let us first focus on compiling programs without communication and alternation. Circuits generated by our compiler can be classified into three types: construct block, register block, and wiring block, to be laid out according to the following convention:



The construct block contains hardware for evaluating expressions, and for implementing the token-passing control scheme. The register block contains D-type flipflops for implementing the variables in a user program. It is connected to the construct block through the wiring block, which includes broadcast and multiplexing circuitry for reading from and writing to a register.

For simplicity, the req input for each construct block enters on the left-hand side, while the ack output emerges from its right. The *starter* block, which produces the initial token, is placed on the left of the main program P (Figure 2).

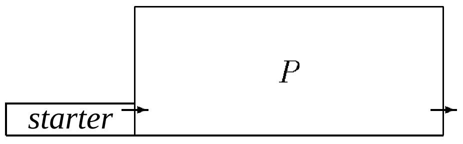


Figure 2 The overall layout for a user program P with the *starter* block. The arrows indicate the location of the req and ack signals for P.

Consider the variable declaration VAR V : Q. The variable introduces a register block and the associated wiring block, shifting the construct block Q upwards (Figure 3). The arrow in the wiring block from V to Q corresponds to data wires carrying the value of registers in V to hardware operators in Q, and the opposite arrow corresponds to wires carrying new values to be stored in the registers. The registers within the register block V can be arranged in many ways; the simplest is to stack them in the order of the variable declaration. v_0 and v_1 are wiring cells for aligning the req and ack signals with those of the neighbouring blocks.

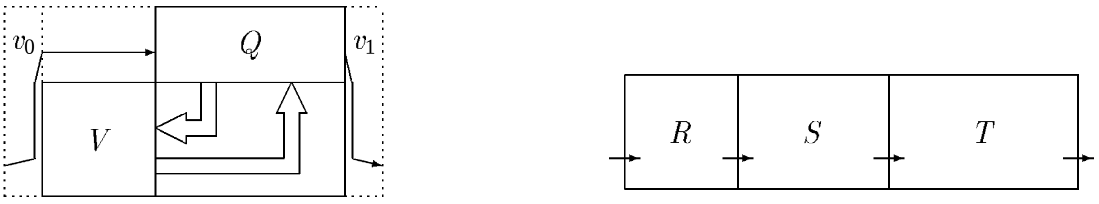


Figure 3 Layout for VAR V : Q and SEQ $R\ S\ T$.

The implementation of SEQ, IF and WHILE are shown in Figures 3, 4 and 5; they obey the above layout convention. Note that the blocks i_0 and i_1 contain respectively the demultiplexer and the or-gate shown in Figure 1(b), and the block w_1 contains both the demultiplexer and the or-gate in Figure 1(c). To avoid complicating the diagrams, data wires are not shown in these figures: for instance if Figure 5 is the Q block in Figure 3, then there may be vertical data wires crossing the horizontal control wires in block w_2 below P, connecting operator hardware in P through the wiring block below Q to the register block V.

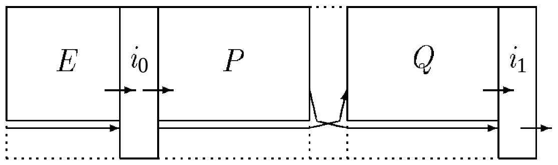


Figure 4 Layout for IF $E\ P$ TRUE Q.

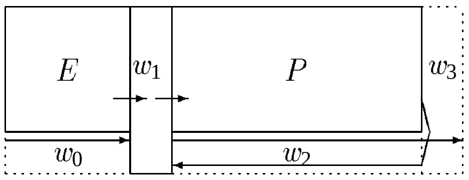


Figure 5 Layout for WHILE $E\ P$.

As an example, the following occam program will be compiled to the circuit shown in Figure 6, with arrows indicating the connections in the wiring block for variable access.

```
VAR x_3 :
SEQ
  x := 0
  VAR y_3 :
  SEQ
    y := 2
    x := x+y
  x:=x+1
```

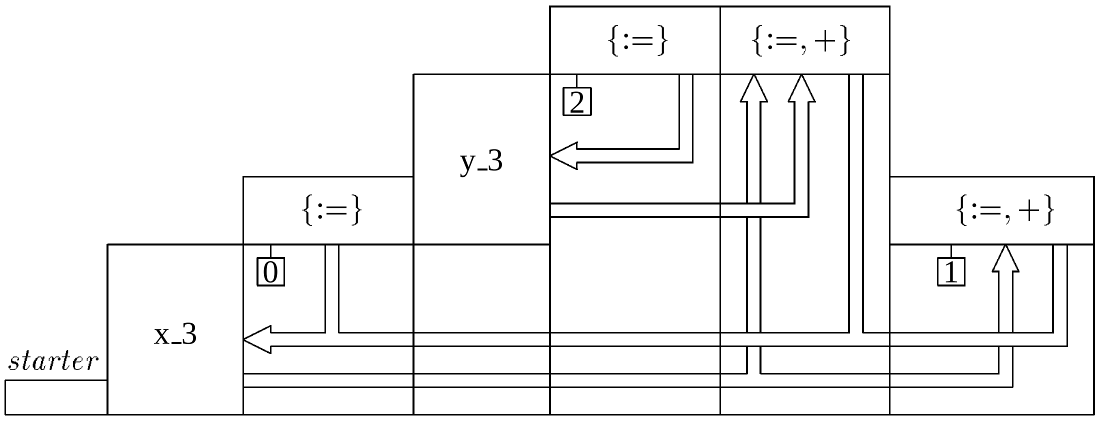


Figure 6 The hardware generated for a program involving two 3-bit variables x and y. Wiring blocks for aligning control signals between neighbouring blocks are not shown.

There are, of course, many ways of laying out a particular occam statement. Our choice is guided largely by simplicity: for instance, placing the two branches of the IF statement side by side (Figure 4) avoids running data wires through operator hardware, but may result in a long, thin circuit. This potential inefficiency can be overcome by having several alternatives for laying out each statement, so that the best can be selected according to the technology and the application.

COMPILER SPECIFICATION AND PROTOTYPE

The compilation scheme outlined in the preceding section can be described by $\mathcal{CP}$, a function from the source language to the OAL language. OAL is a simple language for specifying the relative placement of components; its semantics has been presented elsewhere (Luk and Page 1991), and here we shall illustrate its use in specifying our layout convention.

There are only two constructors in OAL that we require: $beside$ and $below$. The composite circuit with three components P, Q and R having compatible interfaces lying side by side is described by $beside\ [P, Q, R]$; if P is placed below Q and Q is placed below R, then the resulting design is given by $below\ [P, Q, R]$. The compilation function $\mathcal{CP}$ is expressed in terms of the function $\mathcal{C}$, which maps a program P and an initial state ψ to P', the OAL implementation of P, and a new state ψ_P:

$$\mathcal{CP}\ P \quad = \quad beside\ [\textit{starter}, P']$$

where $(\psi_P, P') = \mathcal{C}\ \psi\ P$. The state typically contains information about usage of variables and channels, as well as implementation-specific parameters such as the size and location of control and data signals.

The function $\mathcal{C}$ provides a concise way of specifying our layout convention for occam statements; for instance, the compilation of SEQ (Figure 3) can be described by

$$\mathcal{C}\ \psi\ (\text{SEQ}\ R\ S) \quad = \quad (\psi_S,\ beside\ [R', S']),$$

where $(\psi_R, R') = \mathcal{C}\ \psi\ R$ and $(\psi_S, S') = \mathcal{C}\ \psi_R\ S$.

As another example, given that the function $\mathcal{E}$ maps a state ψ and an expression E to its OAL implementation E' and a new state ψ_E, compiling WHILE (Figure 5) can be expressed as

$$\mathcal{C}\ \psi\ (\text{WHILE}\ E\ P) \;\;=\;\; (\psi'_P,\ beside\ [below\ [w_0, E'],\ w_1,\ below\ [w_2, P'],\ w_3])$$

such that $(\psi_E, E') = \mathcal{E}\ \psi'\ E$ and $(\psi_P, P') = \mathcal{C}\ \psi'_E\ P$, where ψ', ψ'_E and ψ'_P are new states taking into account the effect of the parametrised control and wiring blocks w_0, w_1, w_2 and w_3. The compilation of other statements can be treated in a similar way.

We can use $\mathcal{C}$ and $\mathcal{E}$ to capture alternative arrangements for implementing a particular statement or expression; this will enable the compiler to select the most appropriate arrangement from a number of possibilities, depending on performance or compactness considerations.

While the proposed compilation scheme is device-independent, to demonstrate its viability a prototype compiler has been written for Algotronix's CAL1024 FPGAs (Algotronix 1990). The main tasks include customising the $\mathcal{CP}$, $\mathcal{C}$ and $\mathcal{E}$ functions according to the CAL architecture, and developing CAL-specific libraries in OAL for data and control modules (Luk and Page 1991). Figure 7 shows the CAL circuit produced by our prototype compiler for the program

VAR x_1, y_1 : SEQ x:=0 y:=1 x:=x+y

Note that latch 1 is part of the control hardware for x := 0, since expression evaluation is assumed to complete in one cycle. Similarly latch 2 and 3 control the other two assignments.

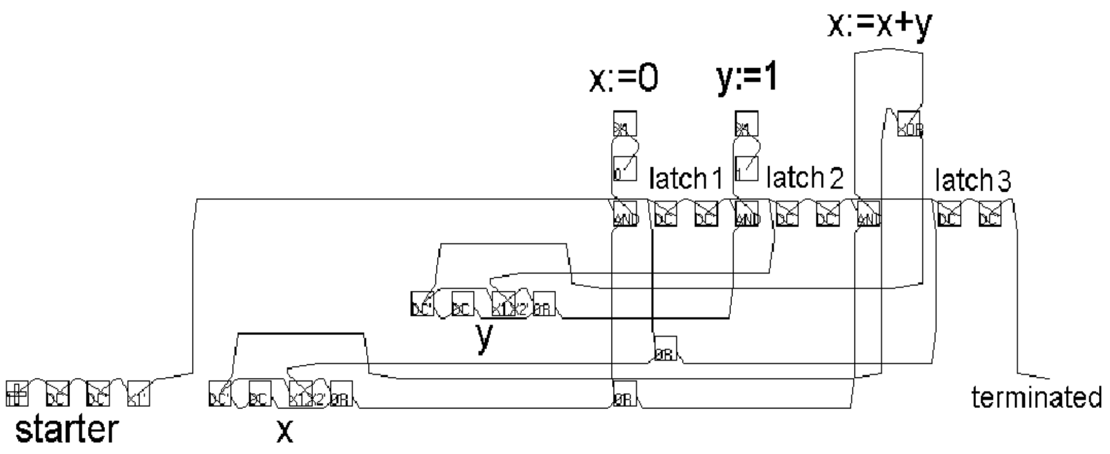


Figure 7 Algotronix layout for a simple program involving two 1-bit variables x and y.

COMPILING COMMUNICATION STATEMENTS

Two parallel processes in occam can communicate with each other through a channel. The command $C!E$ sends the value of the expression E down the channel C; the command $C?X$ receives a value from the channel C and stores it in the variable X. Channel communication can be regarded as distributed assignment: the program fragment

CHAN C : PAR $C!E$ $C?X$

behaves the same as X := E.

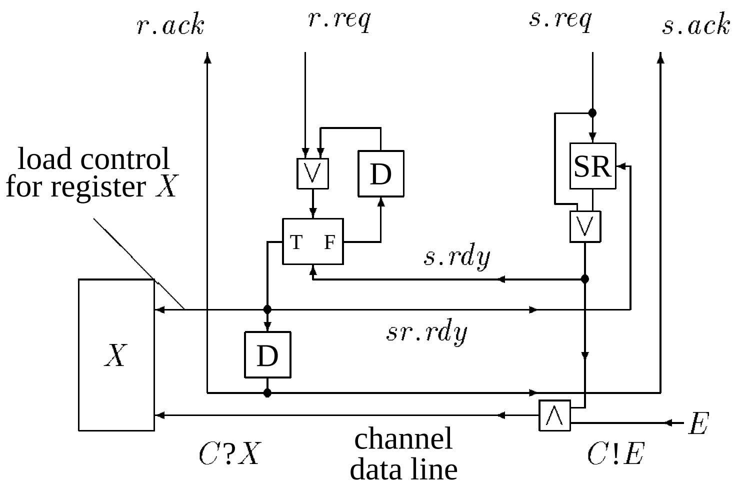


Figure 8 Hardware for $C?X$ and $C!E$.

The hardware for implementing channel communication is shown in Figure 8. The $s.rdy$ (sender ready) signal controls a demultiplexer on the receiver side, so that after the token has arrived at $s.req$, the req input for the sender, the or-gate and the (reset-dominant) SR-flipflop maintain the $s.rdy$ wire high until the receiver is also ready. If the receiver obtains its token before the sender does, the token will be stored in a D-type flipflop until the sender is ready. When both sides are ready, a pulse will emerge from the T-output of the demultiplexer which has three effects: (a) it loads the value E into the register holding the value of the variable X; (b) it resets the SR-flipflop in the sender through the $sr.rdy$ (sender and receiver ready) wire; and (c) it provides the acknowledgement signal for the sender and the receiver after one cycle delay (which corresponds to the delay associated with the assignment statement). The and-gate in the sender is part of a multiplexing scheme to allow the same channel to be used more than once – similar hardware is used to control the storage of different values in a variable.

The layout for two communicating processes is shown in Figure 9. When a channel is declared, the part of the circuit for which it is valid is shifted upwards to leave space in the wiring block for connecting the sending process to the receiving process (cf. the layout for variable declaration in Figure 3).

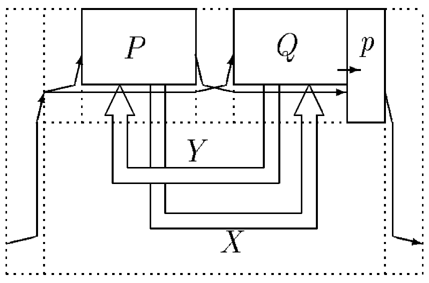


Figure 9 Layout for CHAN X, Y : PAR P Q, where X is a channel from P to Q and Y is a channel from Q to P. The block p contains the control hardware for PAR.

Channels can also be used to interface a piece of hardware to external circuitry, as long as both sides agree on a common protocol. Our prototype compiler has two external channels, ext.in and ext.out, and the environment is assumed to be always ready to send or to receive data. The following is a program for run-length coding, which repeatedly accepts 4-bit values from ext.in until a new value is received; it then outputs the value and the number of occurrences modulo 16 to ext.out. This run-length coder is initialised to look for zero's, hence the sequence of inputs 1,1,1,3,4,4,4,4,4,4,7,7,1 from channel ext.in will produce the sequence of outputs 0,0,1,3,3,1,4,6,7,2,1,1 on channel ext.out.

```
CHAN ext.in_4, ext.out_4 :
VAR current_4, count_4, previous_4 :
SEQ
  count := 0
  previous := 0
  WHILE TRUE
    SEQ
      ext.in ? current
      IF
        current = previous
          count := count + 1
        TRUE
          SEQ
            ext.out ! previous
            ext.out ! count
            count := 1
            previous := current
```

The result of compiling this program can be seen in Figure 10. Note that `IN0`...`IN3` carry input data for ext.in, `OUT0`...`OUT3` carry output data for ext.out, `OUT4` is $sr.rdy$ for ext.in, `OUT5` is $s.rdy$ for ext.out, and `OUT6` provides the ack signal for the program.

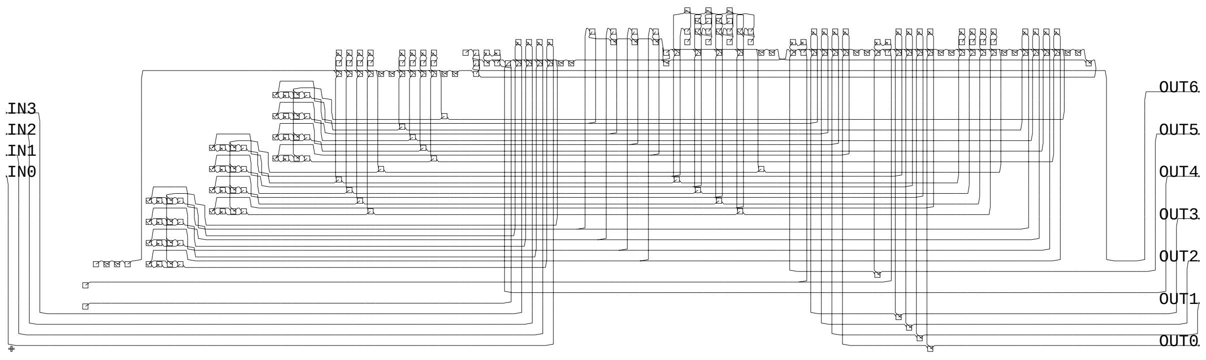


Figure 10 Algotronix layout for a program that performs run-length coding.

COMPILING PRIORITISED ALTERNATION

The prioritised alternation statement is similar to the conditional, except that the choice may depend on the order of arrival of messages from one or more senders. For instance, the statement

$$\text{PRIALT} \quad (E1\,\&\,C1?X1)\;\; P1 \;\; (E2\,\&\,C2?X2)\;\; P2$$

waits until either $C1$ or $C2$ is ready for communication and the associated boolean expression evaluates to TRUE. If $C1$ is ready first and $E1$ evaluates to TRUE, then $P1$ will be selected; otherwise if $C2$ is ready first and $E2$ evaluates to TRUE, then $P2$ will be selected. If both $C1$ and $C2$ are ready at the same time and both $E1$ and $E2$ evaluate to TRUE, then $P1$ will be selected. The statement terminates when the process selected has terminated.

Figure 11 shows how the alternation statement can be implemented in hardware. The function of the demultiplexer is similar to that in the receiver (Figure 8), and the D-type flipflop stores the token until it is released to either $P1$ or $P2$. One possible layout of this circuit is shown in Figure 12. In this figure, the cell a_1 contains the or-gate connected to the req input and the control circuitry for checking $E1$ & $C1?X1$, while a_2 contains the D-type flipflop for storing the token and the control circuitry for checking $E2$ & $C2?X2$. The cell a_3 contains the or-gate which provides the ack signal.

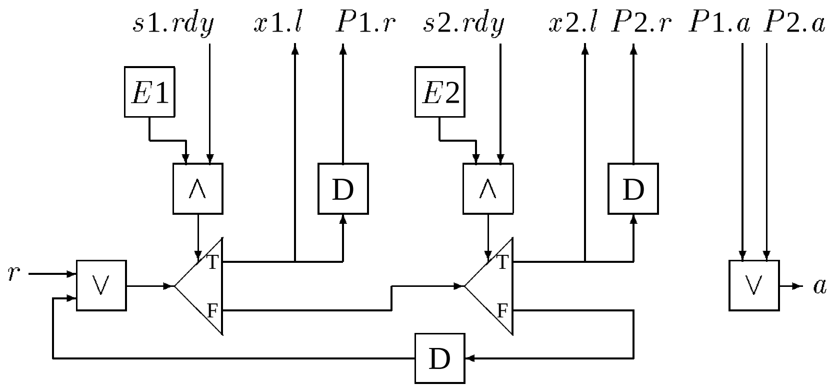


Figure 11 Hardware for implementing prioritised alternation. req is abbreviated to r, ack to a and $load$ to l.

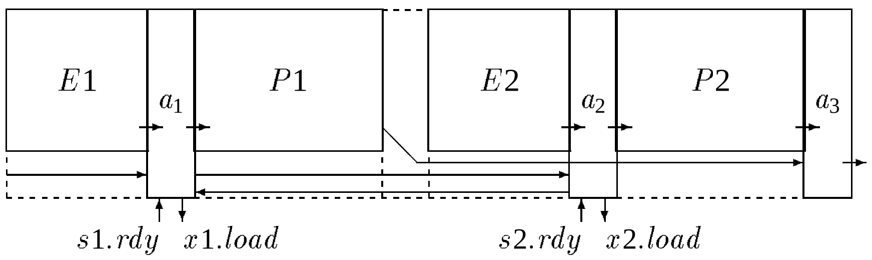


Figure 12 Layout for prioritised alternation.

ESTIMATING SIZE AND PERFORMANCE

To facilitate the selection of the best compiled circuit from a range of alternatives, techniques are included below for estimating the size and the number of clock cycles required for a given program. We hope to extend our approach to cover other properties, such as the maximum clock speed of the compiled circuit, or the time to reach a particular statement in a program.

Let $\mathcal{L}$ be a function that maps an OAL expression A to a pair $(\mathcal{L}_w\, A, \mathcal{L}_h\, A)$, such that $\mathcal{L}_w\, A$ is the width of the circuit for A and $\mathcal{L}_h\, A$ is its height. Given that $\sqcup$ returns the maximum of two numbers (for example $3 \sqcup 5 = 5 \sqcup 2 = 5$), it can be shown that

$$\begin{aligned}\mathcal{L}\,(beside\,[A,B]) &= (\mathcal{L}_w\, A \,+\, \mathcal{L}_w\, B,\; \mathcal{L}_h\, A \,\sqcup\, \mathcal{L}_h\, B),\\ \mathcal{L}\,(below\,[A,B]) &= (\mathcal{L}_w\, A \,\sqcup\, \mathcal{L}_w\, B,\; \mathcal{L}_h\, A \,+\, \mathcal{L}_h\, B).\end{aligned}$$

Let P' be the OAL expression resulting from compiling P, and $\mathcal{S}$ be a function that maps an occam statement to the dimensions of its compiled circuit; in other words,

$$\mathcal{S}\,P \;=\; (\mathcal{S}_w\,P,\, \mathcal{S}_h\,P) \;=\; \mathcal{L}\,P' \;=\; (\mathcal{L}_w\,P',\, \mathcal{L}_h\,P').$$

A recursive definition for $\mathcal{S}$ can be obtained using the above identities for $\mathcal{L}$. For instance:

$$\begin{aligned}\mathcal{S}\,(\text{SEQ}\;P\;Q) &= \mathcal{L}\,(beside\,[P', Q'])\\ &= (\mathcal{L}_w\,P' \,+\, \mathcal{L}_w\,Q',\, \mathcal{L}_h\,P' \,\sqcup\, \mathcal{L}_h\,Q')\\ &= (\mathcal{S}_w\,P \,+\, \mathcal{S}_w\,Q,\, \mathcal{S}_h\,P \,\sqcup\, \mathcal{S}_h\,Q). \qquad (1)\end{aligned}$$

Other statements in our source language can be treated in a similar way; for example,

$$\mathcal{S}\,(\text{VAR}\;V : Q) \;=\; (\mathcal{L}_w\,v_0 + \mathcal{S}_w\;V + \mathcal{S}_w\;Q + \mathcal{L}_w\,v_1,\; \mathcal{S}_h\;V + \mathcal{S}_h\;Q), \qquad (2)$$

where $\mathcal{L}_w\,v_0$ and $\mathcal{L}_w\,v_1$ are the widths of the peripheral wiring cells (Figure 3) and are equal to one for Algotronix CAL1024 architecture.

If C is a constant, V, X and Y are variables and $\oplus$ is a binary operator, then

$$\begin{aligned}\mathcal{S}\;(V := C) &= (cw \times \mathcal{N}\;V + sa,\; ch) \qquad (3)\\ \mathcal{S}\;(V := X \oplus Y) &= (ow \times \mathcal{N}\;V + sa,\; oh), \qquad (4)\end{aligned}$$

where cw and ch are the dimensions of a one-bit constant generator for C, and ow and oh are the dimensions of a one-bit hardware unit for $\oplus$. $\mathcal{N}\;V$ returns the number of bits in V, and sa is the width of the control hardware for the assignment statement. For Algotronix CAL1024 devices, $cw = 1$, $ch = 3$, $ow = 2$, $oh = 5$ (for the add operator) and $sa = 2$.

The size of a program is given by the function $\mathcal{SP}$:

$$\mathcal{SP}\;P \;=\; (\mathcal{L}_w\;\textit{starter} + \mathcal{S}_w\;P,\; \mathcal{L}_h\;\textit{starter} \sqcup \mathcal{S}_h\;P). \qquad (5)$$

Given that $\mathcal{L}$ *starter*= (4,1) and $\mathcal{S}$ x_1 $=$ $\mathcal{S}$ y_1 $= (5,2)$ for CAL1024 devices, the size of

VAR x_1, y_1 : SEQ x:=0 y:=1 x:=x+y

(Figure 7) can be calculated as follows:

$$
\begin{array}{rrcll}
 & \mathcal{N}\,\mathrm{x} & = & 1 & (a) \\
 & \mathcal{N}\,\mathrm{y} & = & 1 & (b) \\
(a),(3) \Rightarrow & \mathcal{S}\ (\mathrm{x{:=}0}) & = & (1\times 1+2,3)=(3,3) & (c) \\
(b),(3) \Rightarrow & \mathcal{S}\ (\mathrm{y{:=}1}) & = & (1\times 1+2,3)=(3,3) & (d) \\
(a),(b),(4) \Rightarrow & \mathcal{S}\ (\mathrm{x{:=}x{+}y}) & = & (2\times 1+2,5)=(4,5) & (e) \\
(c),(d),(e),(1) \Rightarrow & \mathcal{S}\ (\mathrm{SEQ}\ldots) & = & (3+3+4,3\sqcup 3\sqcup 5)=(10,5) & (f) \\
(a),(b),(f),(2) \Rightarrow & \mathcal{S}\ (\mathrm{VAR}\ldots) & = & (2+10+10,4+5)=(22,9) & (g) \\
(g),(5) \Rightarrow & \mathcal{SP}\ (\mathrm{VAR}\ \ldots) & = & (4+22,1\sqcup 9)=(26,9). &
\end{array}
$$

Hence the compiled circuit is an array of 26 by 9 CAL cells.

As far as performance is concerned, one can use the function $\mathcal{TP}$ to predict the number of clock cycles to complete the execution of a program:

$$\mathcal{TP}\ P \;=\; \mathcal{T}\ starter \,+\, \mathcal{T}\ P$$

where $\mathcal{T}$ *starter* is the number of cycles that the *starter* block requires for initialisation. It should be clear that $\mathcal{T}$ is independent of the layout, and is given by the following equations:

$$
\begin{array}{rcll}
\mathcal{T}\ (\mathrm{SEQ}\,P\,Q) & = & \mathcal{T}\,P \,+\, \mathcal{T}\,Q, & (6) \\
\mathcal{T}\ (\mathrm{PAR}\,P\,Q) & = & \mathcal{T}\,P \sqcup \mathcal{T}\,Q, & (7) \\
\mathcal{T}\ (\mathrm{VAR}\,V : Q) & = & \mathcal{T}\,Q, & (8) \\
\mathcal{T}\ (X := E) & = & 1, & (9) \\
\mathcal{T}\ (\mathrm{IF\ \ TRUE\ \ THEN}\ \ P\ \ \mathrm{TRUE}\ \ Q) & = & \mathcal{T}\,P, & (10) \\
\mathcal{T}\ (\mathrm{IF\ \ FALSE\ \ THEN}\ \ P\ \ \mathrm{TRUE}\ \ Q) & = & \mathcal{T}\,Q, & (11) \\
\mathcal{T}\ (\mathrm{IF}\ \ E\ \ \mathrm{THEN}\ \ P\ \ \mathrm{TRUE}\ \ Q) & = & \mathcal{T}\,P \sqcup \mathcal{T}\,Q. & (12)
\end{array}
$$

Often one can only provide a crude bound for programs involving iteration and communication. For instance, one can define $\mathcal{T}_{max}$ and $\mathcal{T}_{min}$ such that $\mathcal{T}_{max}$ (WHILE FALSE P) $= 0$ and $\mathcal{T}_{max}$ (WHILE $E\,P$) $= \infty$, and $\mathcal{T}_{min}$ (WHILE TRUE P) $= \infty$ and $\mathcal{T}_{min}$ (WHILE E P) $= 0$. A more elaborate analysis would require more complicated rules, like

$$\mathcal{T}_{max}\,(\mathrm{SEQ\ \ i := n\ \ \ (WHILE\ \ i{<>}0\ \ (SEQ}\ \ P\ \ \ \mathrm{i := i{-}1})))\;=\;1+\mathrm{n}\times(\mathcal{T}_{max}\,P+1),$$

provided that P does not alter the value of i.

Program transformation may sometimes help: let A be the program

$$\mathrm{CHAN}\ C : \mathrm{PAR}\ \ (\mathrm{SEQ}\ \ P\ \ C!E)\ \ (\mathrm{SEQ}\ \ Q\ \ C?X)$$

where P and Q do not mention C. Since communication occurs when both the sender and the receiver are ready, A can be rewritten as SEQ (PAR P Q) (CHAN C : PAR $C!E$ $C?X$) which, upon replacing communication by assignment, becomes SEQ (PAR P Q) ($X := E$). Equations (6), (7) and (9) then give $\mathcal{T}\,A = (\mathcal{T}\,P \sqcup \mathcal{T}\,Q) + 1$.

SUMMARY

Placement and routing takes up a significant proportion of the time in implementing designs in hardware, especially for many field-programmable devices. Experience in computing has indicated that the structure of a high-level program is the key to understanding its behaviour; this paper shows that the program structure may also be used to guide the layout process, and to estimate the size and performance of the compiled circuit.

Our efforts have been concentrated on a framework for mapping programs into layout, and we have not made any attempt to optimise the compilation scheme or the prototype compiler. To become a practical tool, our system can be extended in a number of ways, such as including a mechanism – deterministic or otherwise – for selecting an appropriate strategy to lay out each statement in the source program. Further layout optimisation using methods like min-cut or simulated annealing, as well as device-specific compaction techniques, should also be studied. For instance, systematically compacting the design in Figure 7 by hand reduces its size from 234 cells to 64 cells, and its critical path from 132 ns to 80 ns, while repeatedly applying Algotronix tools based on simulated annealing yields 40 cells and 54 ns. We hope to automate the hierarchical application of these techniques to provide an incremental route for optimising designs. Moreover, since the mapping from program statements to hardware is known, it would be possible to adopt different optimisation methods for different parts of the layout.

Given that the proposed compilation scheme is based on three kinds of blocks, each fulfilling a particular function, an efficient implementation may involve a mixture of devices. As an example, fast memories and a small FPGA may be employed for the register block, programmable crossbar or field-programmable interconnect chips for the wiring block, and large FPGAs and dedicated hardware for the construct block.

Source transformation (Roscoe and Hoare 1988) can also be adopted for optimisation and for performance analysis. For instance, it may be possible to minimise the number of variables in a program, or to optimise the amount of parallelism and pipelining for a specific application.

Our approach is distinguished by the use of OAL, which provides a concise means of capturing layout and an interface between the device-independent compiler and the device-dependent circuit generators. Further work will explore how OAL can assist in verifying the correctness of our compiler: assigning hardware to programs while preserving their behaviour.

ACKNOWLEDGEMENTS

Thanks to Geoffrey Brown, Jim Davies, Tony Hoare, Tom Kean, John O'Leary and Mike Spivey for inspiring discussions and LaTeX advice, and to ESPRIT OMI/MAP and OMI/HORN Programmes and Algotronix Limited for supporting our work.

REFERENCES

Algotronix Limited, *CAL 1024 Datasheet,* 1990.

Inmos Limited, *The Occam Programming Manual,* Prentice Hall, 1984.

Luk, W. and Page, I., "Parameterising designs for FPGAs," in *FPGAs,* W. Moore and W. Luk, Eds., Abingdon EE&CS Books, pp. 284–295, 1991.

Page, I. and Luk, W., "Compiling occam into FPGAs," in *FPGAs,* W. Moore and W. Luk, Eds., Abingdon EE&CS Books, pp. 271–283, 1991.

Roscoe, A.W. and Hoare, C.A.R., "The laws of occam programming," *Theoretical Computer Science,* vol. 60, pp. 177–229, 1988.

Weber, S., Bloom, B. and Brown, G.M., "Compiling Joy into silicon," *Proc. Brown/MIT VLSI Conference,* MIT Press, pp. 79-98, 1992.