# Computing Lab Technical Report No. 10/92
# A Video Control Processor

G. E. W. Tripp

University of Kent at Canterbury

ABSTRACT

This paper describes the design and implementation of a simple microprogrammable processor with a fast response time to external events. This processor was designed as a controller for a testbench video system capable of performing a number of functions such as windowing and frame rate conversion.

## 1. Introduction

The design of the processor described in this paper was part of a project called the video server project. The aim of this project was to design a system that would allow the display of video windows on the screens of workstations. This type of design was slightly unusual in that it was based on a client server model of interaction. In this case the workstation(s) being the client and the system designed in the project being the (video) server. The role of the video server was to be able to service one of a number of workstations.

The workstation(s), the video server and a number of video sources were all connected to an analogue video circuit switch network. This network was relatively straight forward to build and acts as a substitute for the fast digital networks that will eventually be available.

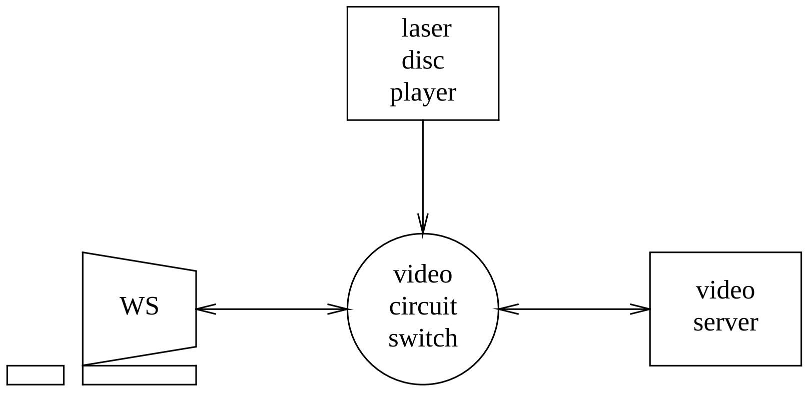

Figure 1.1 - Video Network

### 1.1. The Video Server

The aim of the video server was to be able to take input from the analogue video network in the form of broadcast standard 625 line 50 Hz interlaced video and save this in an internal frame store. The video server would then use the video being saved in the frame store to generate video output at a scan rate suitable for high resolution workstations.

The video server would be used by a client workstation to generate a video window overlay to be displayed on its screen.

To use this system, the workstation would need to provide the video server with the size and clipping information for the video window to be displayed and also ensure that the corresponding area on the screen is left blank.

To enable the output from the video server to be displayed on the screen of the workstation, the workstation would supply a copy of its video sync signal for the video server to lock onto and mix the video output from the video server with the output from the workstation itself before sending the composite signal to the screen.

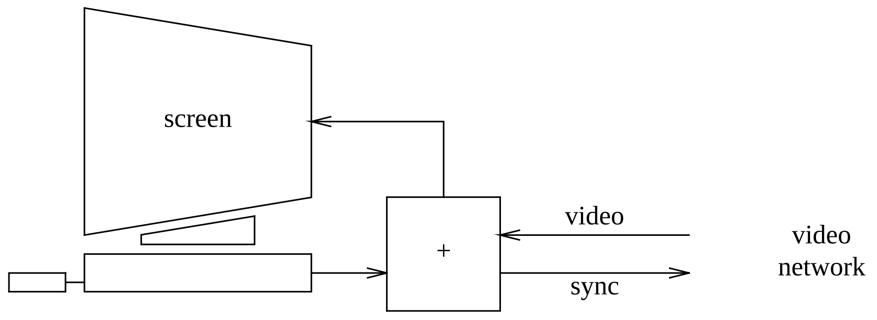

Figure 1.2 - Connection of workstation to video network

## 1.2. Proposed method of operation

It was decided early on that if such a system was to be designed, then it should be as flexible as possible and not just display a fixed size video window. Instead of working with a fixed block of video, the video server would define how the video was to be saved and regenerated by the use of a number of control structures. This method would enable software control over the display and capture of video data.

The unit of data to be handled by this system was to be the line segment. This was defined as a consecutive vector of pixels on a given scan line. The control structures would contain references to the line segments stored in the frame store and would enable any input line segment to be displayed in any position on the video output.

The system of control structures chosen for the video server consisted of a linked list of frame control structures for each of video input and video output. Each of these frame control structures has a pointer to a linked list of line control structures that define the active scan lines. The line control structures define the starting position in memory to be used for the storage or retrieval of video data.

The lines of incoming video can be stored in the video frame store as specified by the controlling system. Each line is stored in the frame store as defined by the line control structure for that line, with consecutive pixels saved in increasing or decreasing address order.

For video output, a vertical offset in the frame control structure defines how far down the screen the video starts and a vertical height defines how many lines are to be displayed. For each line of video that is displayed, the line control structure contains a horizontal offset that defines how far from the left edge of the screen the video is to start and a horizontal width that defines the number of pixels to be displayed.

## 1.3. The design of the video plane

To perform this list of operations, a number of demands are made of the system to be built. A large amount of the work consists of video input and output from a video frame store memory. This part of the design was considered first.

A high output data rate is required to generate video for a workstation screen. This is of the order of 100 M pixels/sec for a number of current systems. This type of data rate has been difficult to achieve in the past, because of the relatively slow speed of large memory chips. This is normally overcome by the use of wide blocks of memory, with each word holding a number of pixels - for example with 100ns memory we would need at least 10 pixels wide store to achieve the data rate of 100 pixels/sec.

This problem has been overcome by the use of a new type of memory called video DRAM [1]. This is effectively dual port memory with a standard random access port and also a serial port. The memory chips contain an internal shift register, which is connected to the serial port. Data can be clocked into or out of the shift register at a relatively high speed (25 MHz). The shift register can be loaded/saved from/to a complete row of the internal memory array. This type of chip is designed for use in video output stages of workstations as it enables the screen to be refreshed with a minimum of overhead. The internal shift register is loaded at the start of a scan line and the contents of the register clocked out during the active part of the scan line to generate the video data. For the video server project, it was possible to build a video memory array of only 4 pixels wide to be able to generate video output data at the rate of 100 M pixels/sec. The pixels within the system were 8 bits wide giving a store width of 32 bits.

This type of memory requires little overhead for generating the video output, but is rather difficult to drive. The memory chips need to be clocked at the correct time and the transfer of data within the chips from memory array to shift register needs to be done within a relatively short window during the inactive part of the line scan. The memory driving is normally done using a special video memory controller chip. This type of chip will perform all the internal transfer cycles, handle memory refresh and also provide a user port into the memory. However, at the time of this design, these chips were only capable of displaying a fixed block of memory on the video output - whereas with this project, there was a requirement to be able to select individual line addresses and screen position in software. There appeared to be no special purpose chip that could be used to perform the functions required by the project, so alternative methods of driving the memory were investigated.

The use of the video DRAM was determined by the requirement to support high output data rates. The data rate requirements for input and output were asymmetrical in that around 100 M pixels/sec were required on the video output whereas only 10 - 20 M pixels/sec were required for the video input. It was therefore an obvious choice to use the serial port of the video DRAM chips for video output and use the conventional random access port for video input and all other functions.

It was decided initially that a full colour system would be desirable but not a pre-requisite. Because of this, the video memory system was designed as a single colour plane with the ability to add other planes later if required. The use of multiple colour planes implied that some method of synchronisation was required across the different planes - to keep them all operating together. Because of this, it became apparent that individual control and timing systems within each colour plane would be unsuitable. It would be better to have a global control system that would keep all colour planes operating together in sync. It was decided that the control system should be removed completely from the video colour planes. The video (colour) planes would contain only the circuitry needed to sample and regenerate video information, the video memory for that plane and some general control and timing functions. The video plane cards would then be plugged into a bus or backplane that would supply all the control and timing functions remotely.

All data for video input and output would be kept within the video plane cards themselves (so as not to limit the performance due to bus bandwidth) and commands provided over the bus to instruct the video plane modules what function they should perform.

Each video plane would consist of two separate memory planes, an 8 bit video plane to hold the current video image and a 4 bit attribute plane to hold clipping and blanking information. The video input would be written into the video plane on a continuous basis. The video output would read data from both video and attribute plane and use the attribute plane to decide whether to output the video data directly or whether to perform some type of blanking or highlighting operation on it first. The attribute plane enables the video server to produce video windows with any required clipping pattern. When the clipping information changes, the attribute plane would need to be updated to reflect this.

At this early stage, there was a distinct split between the functions performed within the system. The video planes would each handle a single colour signal and all control and timing operations would be performed elsewhere. The control of the video planes would be done over a synchronous bus called the video plane bus. The functions performed would be determined by a 16 bit command word on the bus that would determine the operation performed on the video plane cards during the next bus cycle. A 32 bit general purpose data bus was also provided to enable memory addresses or video data to be transferred between the control module and the video plane module(s).

All clocks for video sampling and regeneration were also to be provided externally.

The interface to the bus was defined and the video plane modules were designed. The assumption here was that the control functions would be provided by some method to be determined later. The functions performed by the video plane memory cards are described elsewhere [2] and little detail of the operations will be given here.

### 1.4. The control system and its requirements

When designing the video planes, the control system was treated simply as a black box that performed all of the required control functions. Later when designing the control system, thought needed to be given on how to meet these requirements.

A number of different operations were needed from the controller. Some of these were to do with the video sync systems and window timing and others were to do with the management of the memory.

The sync signals from the video input needed to be decoded so as to sample the video input in the correct position and a video input clock needed to be generated to do this sampling. This input video clock would be provided to each of the video plane cards. The video plane card would sample video data coming in and save it in an input FIFO. When the FIFO contained a certain amount of data, the contents would need to be flushed to the video frame store. For each video input line, the correct position in memory would need to be determined for the video data.

On the video output, the incoming sync signal from the workstation needed to be used to control the timing of the video output signals and to enable the video output clock to be generated at the correct times. At the start of each scan line, the video DRAM would need to be set up for the current scan line to enable video output to begin from the start address defined by the line control structure for that line.

As well as these I/O operations, the controller would need to ensure that the memory is regularly refreshed and enable the contents of the video planes to be read and written. This latter operation would also be used for the setting up of window clipping information in the video frame store by writing to the attribute plane.

### 1.5. Partition of functions

The functions mentioned in the previous section tend to fall into a number of categories. There are some functions that are obviously only possible if implemented directly in hardware - such as sync separation, video timing and clock generation. Others could be implemented in software - such as the generation of frame and line control structures to control the size of windows and the use of memory. Some of these functions however are real time operations that could be implemented in software only if the strict crisis times are met. Examples of these real time operations include: the refreshing of the memory that has to be done 256 times each 4ms; copying video input data from the input FIFO to the video frame store, which needs to be done before the FIFO overflows (holds 64 pixels) and worst of all the setting up of the video DRAM for video output that has to be done during the line flyback time (about 4ms).

Several possible methods were considered for implementing these operations and these are covered in the next section.

## 2. Study of possible implementation methods

It was evident that a number of functions needed to be performed, each having different characteristics and requirements. Some of these functions were complex to perform but had relatively relaxed time constraints. Some functions were quite simple but had short crisis times. Finally, some functions were simple hardware timing functions.

The window sizing and clipping algorithms would be relatively complex. These could be implemented easily in software in a high level language, but would be almost impossible to implement directly in hardware.

The bus and memory handling functions could be implemented in hardware or by some type of simple sequencer. It is unlikely that a standard microprocessor would be able to meet the short crisis times imposed by this system. If this function was implemented in hardware, it would be quite complex and would also be relatively inflexible. With the control of windows and memory addresses, it is also important to ensure that the control structures used to define window geometry and video memory addresses are available for use when required.

The window timing functions are easy to implement in hardware and it would be inappropriate to implement these in any other way.

## 2.1. First Solution

The original idea for controlling this system consisted of having a simple sequencing system driving the video plane bus and with the main window clipping and sizing functions being performed by a microprocessor system such as an M68000 [3]. The video memory addressing information and window size information would then be fetched from the M68000 memory using DMA.

This was the original framework for the project and remained so while the video plane cards were being designed. Later, when the control system was being designed, this solution did not appear to be so attractive. The M68000 system with multiple cards in a rack did not look to be the most cost effective way of providing processing power. Also, with this original approach the main processor bus would be busy for much of the time with handling DMA requests from the video plane bus controller for frame and line control structures. There would also be no guarantee of immediate access to the M68000 bus, so data would probably need to be pre-fetched before needed or even kept in FIFO storage.

## 2.2. The Second Solution

The second idea was to use a transputer as the main processor and to put this on the same card as the video plane bus controller. This system would have shared memory between the bus controller and the transputer. This would probably be the ideal method of implementation, except for the complexity of the hardware that would have been created. The shared memory interface between the transputer and the bus controller looked like being rather messy and there was still a need for the ability to pass events between the two systems. The passing of events was solved by the proposed use of the transputer link - with the video plane bus controller interfacing to this via a transputer link adaptor.

## 2.3. The Third Solution

The third solution and the one that was adopted, was to dispense with the on board transputer and to communicate directly with a remote transputer system via a standard transputer link. This transputer link would carry all the information between the two systems and data would be stored in memory local to the video plane bus controller. It was felt that apart from the available data rates given by shared memory, there would be little advantage to having the transputer on board and it would be more cost effective to provide a transputer system elsewhere such as on a PC.

The proposal now was to have a video plane bus controller with local memory and a transputer link to the outside world.

## 2.4. Method adopted

Given the design strategy in the previous section, the next stage was to decide how this system could be structured. The original idea was to put all the control functions onto a single card. This design was however only in the prototype stage and it was likely that modifications may have been required later.

After careful study, it became evident that this controller consisted of two distinct parts: a control processor or sequencer and several pieces of timing logic that were specific to the control of this system. The control processor would be likely to remain static whereas the various pieces of timing logic might vary depending on future upgrades or modifications. These two parts of the system were however very closely linked, in that the controller needed to load values in the timing logic at various times that would normally be specified by the timing logic itself.

At this stage it was decided that the controller would be split into two separate cards: the video controller and the window controller. The video controller being the control processor and the window controller performing all of the timing functions such as: sync recovery, video clock generation and window timing.

The split between these two cards required an interface to be defined between them. This looked to be quite complex because of the close links between these two cards, however all that was needed was to provide a data highway to enable the control processor to write values into registers on the window controller and some type of event interface to enable the window controller to signal events to the control processor.

The interface between these two cards was resolved by plugging both into the existing video plane bus that was intended for the control processor to communicate with the video plane cards. This video plane bus was extended by the provision of a 4 bit command word to enable the control processor to control the functions performed by the window controller and a number of signals for carrying the event signals from the window controller to the control processor. Any data transferred between the two cards would be done via the 32 bit data highway already present.

All of the hardware specific to controlling the video server was now removed from the video controller card, leaving a simpler, tidier piece of circuitry. However, the functions now performed by the video control processor looked like they were to be rather more complex than those originally envisaged. The control processor was now to be responsible for servicing the window controller as well as just controlling the video planes and the video plane bus. The new designs for the control processor were simpler in that no timing functions needed to be provided. All timing events, including memory refresh and deciding when to flush the video input FIFO's would be decided by the window controller and an event generated to request that this operation be performed.

The original ideas for the construction of the video controller were that it would be a simple sequencer built from a number of bit-slice components. These types of designs are often quite appropriate for simple control functions as they are relatively flexible - because of the ability to change their function by updating their microcode. It now looked like there might be rather more functionality required than this type of system could provide.

The design now changed to increase the functionality of this controller into what effectively was a simple processor. The requirements for this system were however rather different to those of a conventional processor. There was no real need for a processor capable of performing complicated functions or operating at a high speed. The complexities of window geometry calculations would be performed by the remote transputer system and there was no need to operate very quickly. What was needed by this system was the ability to respond quickly to events happening within the system and handle these events within their crisis times. This however is not an operation given much priority within most microprocessors.

## 3. Design of the Control Processor

The main aim of the control processor was to be able to handle a number of events of various importance. Some of these events were extremely important in that they needed to be handled within a few microseconds, whereas others needed to be acted upon within a few tens of microseconds or may have no specific crisis time at all.

### 3.1. Prioritised events

The suggestion from this observation was that the events could be sorted into order of importance, with the most important events being acted upon first and the least important events acted upon last. This is a simple operation to perform and is commonly found in standard interrupt controller chips. This type of function can be performed by supplying all the event inputs to a priority encoder that can give the bit number of the most significant active bit.

It is likely that on some occasions, there will be one event being acted upon and then a more important event will take place. In this case, it may be necessary to suspend servicing of the low priority event and service the event of higher priority. This may only be necessary if the low priority task is likely to execute long enough to exceed the crisis time of the higher priority task.

## 3.2. Processor Model Used

For this type of system, the operations performed do not require a full processor to be designed. A processor that just executes microcode directly - rather than executing a full machine code instruction set - is perfectly adequate for this purpose. This type of processor is really a type of micro-programmed controller rather than a conventional CPU.

Admittedly, the microprogrammed controller is more difficult to program than the traditional processor that provides normal machine code execution. However, the execution of machine code requires an op-code fetch execute cycle to be performed with all the associated instruction registers and register selection mechanisms. The machine code execution is also likely to be slower unless some form of pipelining is provided.

The provision of a simple micro-programmed machine implies that the software is going to remain mainly static, because of the difficulty of writing and debugging microcode - but does give far more flexibility than a completely hardwired solution.

## 3.3. Scheduling and Tasks

The use of such a simple machine enables a custom system to be used to handle the external events. On a conventional processor this would normally be done via some type of interrupt mechanism. Microcode handling the interrupts would suspend the current machine code execution, save several registers on the stack and then call the interrupt routine associated with that event.

With this system, there was no particular wish to have to go through a complex interrupt service routine and even less to have to save registers on some type of stack. These operations add time to the interrupt overhead, which is undesirable in this application.

Rather than providing a general interrupt mechanism, each microcode task (or thread) would be responsible for its own de-scheduling and would have its own storage space in which to save its current registers. One method of operation is for each task to de-schedule itself, when it has completed, or when it notices that another task of higher priority is waiting. The task would deschedule itself by saving a copy of its restart address (which might be start of the routine or may be part way through) and then calling a common task dispatcher that would determine the next task to be executed, restore its restart address and start it executing again.

## 3.4. Parallel Task Scheduling

This method of scheduling is quite efficient in that each task is allowed to deschedule at appropriate points, normally when little context needs to be saved - but has the disadvantage that it relies on the tasks descheduling and not hogging the processor. Unfortunately for this application, there are several tasks with short crisis times waiting to use the processor. Because of the short execution time for each task, the task dispatcher can take up a large amount of the available processor time and significantly shorten the crisis times seen.

To improve the performance of this system given the expected load, it was decided to provide some extra hardware support for the scheduling process. Extra storage was provided to hold the context of each of the microcode tasks - in this case only the restart address - and a mechanism was provided to save and restore the restart addresses in parallel with other operations. To ensure that this functions as efficiently as possible, it is necessary to include logic to enable the number of the next, highest priority task to be determined simultaneously.

The outcome of this provision of parallel task scheduling is that it is possible to perform a descheduling operation in parallel with the last microinstruction executed by a task. The scheduling operations are performed during the execution of the final instruction of one task and enable the first instruction for the new task to be executed in the next microcycle. This should imply that there is no overhead at all in rescheduling, but in practice the amount of work taking place during the final instruction of a task means that the time for the last instruction needed to be extended by 50ns.

This rescheduling system has several options and can offer to deschedule while performing a branch operation, or offer to deschedule if a higher priority task is waiting to execute. In some cases, this mode of operation allows single microinstructions from a task to be executed before swapping to a different task. This makes very efficient use of the processor.

## 3.5. Transputer connected via buffered link

The connection to the outside world from this system was to be a standard transputer link. This was buffered to the RS422 specification so that a reasonable distance could be covered between the video server and the PC containing the remote transputer without having any problems with transmission.

This link needed to be handled quite quickly, so this was set up in such a way as to generate hardware events on receiving data from the link or when the last piece of data had been sent. For this purpose two of the hardware event signals were reserved for this purpose on the controller card and another two were also reserved for the idle loop and for a main program if required. These events corresponded to the four lowest priority events in the system.

## 3.6. Overview of architecture

This section gives a short overview of the architecture of the video control processor. More comprehensive implementation details are given in section 4.

### 3.6.1. Data paths

The video control processor is built from standard bit slice components. The ALU section of the processor is constructed from two Am29C101 [4] bit slice ALUs, preceded by a byte (barrel) shifter constructed from PALs.

All data transfer within the processor takes place via a 32 bit data bus called the XD-BUS, which has several sources and destinations. This bus also interfaces to the data highway on the video plane bus to enable the control processor to communicate with the video planes or the window controller.

### 3.6.2. Sequence of execution

The sequencing of instructions is controlled by an Am29C10A sequencer [5] and a certain amount of additional logic.

The sequencer circuitry also contains a set of task address registers that are used to hold the current execution address of any of 16 tasks that are currently suspended.

### 3.6.3. Video Plane Bus Interface

The processor interfaces directly with the video plane bus, providing access to the 32 bit data bus.

It provides a 16 bit command to control the video planes and a 4 bit function to control any other cards (the window controller). These command and function fields are generated (with slight modification) from fields in the microprogram.

An ‘event’ interface is also provided to the Video Plane Bus, this consists of event ready inputs for events 4 to 15 and a 4 bit current event output.

### 3.6.4. The Transputer Link Interface

The only external connection to the video control processor - apart from the video plane bus - is a single transputer link. This transputer link is externally buffered to RS422 levels and is connected to a transputer link adapter on the video control processor itself. The transputer link adapter can generate events to the video control processor that can cause an input and output task to be scheduled.

### 3.6.5. Microprogramming

The video control processor is programmed directly in microcode, with no conventional machine code execution. The microcode is 80 bits wide and held in EPROM on the video control processor card itself. The processor can directly address up to 4K words (or lines) of microcode.

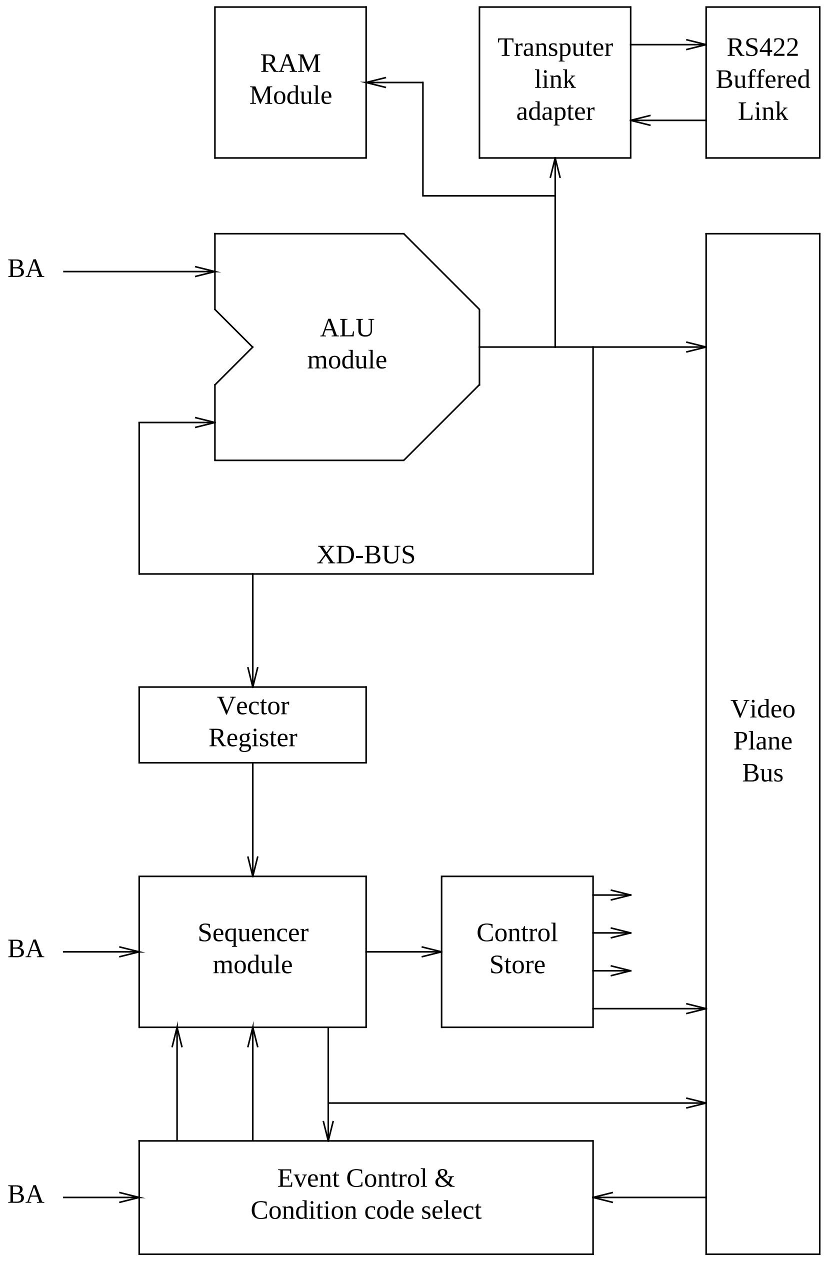

Figure 3.1 - Block Diagram of Video Control Processor

## 3.7. Scheduling

There can be up to 16 tasks running on the Video Control Processor. The tasks are called Task0, Task1 .. Task15. These tasks are assigned a fixed priority, where Task15 is the highest priority and Task0 the lowest.

Each task has an execution state that can be either RUN/HALT, this defines whether the task may be executed. Each task also has event input that may be WAIT/READY which can request that the task be scheduled.

At any scheduling point, the task to be executed is the highest priority task that is in state RUN and also has READY on its event input. Once a task has started running, it continues to do so until it de-schedules itself. In this sense, the scheduling is non preemptive - there is no possibility for a high priority task to ‘interrupt’ a low priority task, but can only take over if rescheduling takes place.

### 3.7.1. Scheduling and de-scheduling

Because of the ability of one task to block out another task of higher priority, the micro-programs need to be written so that no task hogs the processor for too long: as work progresses on this project, it should be possible to define a figure for maximum use. There are several ways in which a process can de-schedule, or offer to do so.

### 3.7.2. Wait for event

When waiting for an event to occur, a task should de-schedule itself by executing a sequencer WAIT operation. This will cause the task to be suspended until the task event input becomes READY. The sequencer WAIT operation is conditional and can be used to de-schedule if the task is not ready.

### 3.7.3. Stop execution

If executing a continuous loop of instructions, a condition code test of HIPRI can be made, which will be true if a higher priority event is waiting. A task can then tidy up any work it is doing and de-schedule itself by executing a sequencer JMAP operation.

### 3.7.4. Offer to de-schedule

If a task is in a convenient point to de-schedule, but would like to carry on if it is still the highest priority task, then it can execute the sequencer NEXT operation. The NEXT operation is used in place of a CJP (conditional jump) operation and will offer to de-schedule. If the task is de- scheduled, then it will continue later at the address specified by the NEXT operation - if it is not de-scheduled, then it does a jump to the address specified.

In any of these cases, if the task is de-scheduled then the only information kept about that task is the task restart address. These addresses are kept in the task address registers and are updated by NEXT or WAIT. JMAP does not update the task address registers, hence allowing a task always to be re-entered at the same address. It is important always to ensure that the stack is empty when de-scheduling as there is only one stack that all tasks share.

### 3.7.5. Use of tasks.

Some of these tasks are reserved for specific purposes, others are available for general use.

TASK0
: This is the NULL task, it is always in state RUN and has a fixed event input of READY. This task will be executed if nothing else is waiting and should normally just de-schedule itself.

TASK1
: This task is effectively the base level task that runs most of the time. It has a fixed event input of READY, but may be put in execution states of RUN or HALT.

TASK2
: This is the transputer link output task. It becomes READY when the transputer link adapter is ready to transmit a piece of data and goes to WAIT when data is written to it. This task can be put into execution states of RUN or HALT.

TASK3
: This is the transputer link input task. It becomes READY when the transputer link adapter has received a piece of data and goes to WAIT when the data is read. This task can be put into execution states of RUN or HALT.

TASK4 .. TASK15
: These tasks are undedicated. The event inputs come from the video plane bus. These tasks can be put into execution states of RUN or HALT.

## 4. Implementation Details

This section gives the implementation details of the video control processor. Some parts of this section refer to the microinstructions used for various operations, little detail is given of the operation of standard bit-slice components and the reader should refer to the manufacturer's data sheets for more details.

### 4.1. The ALU module

The ALU module consists of several components, each of which is controlled separately by fields in the micro-code.

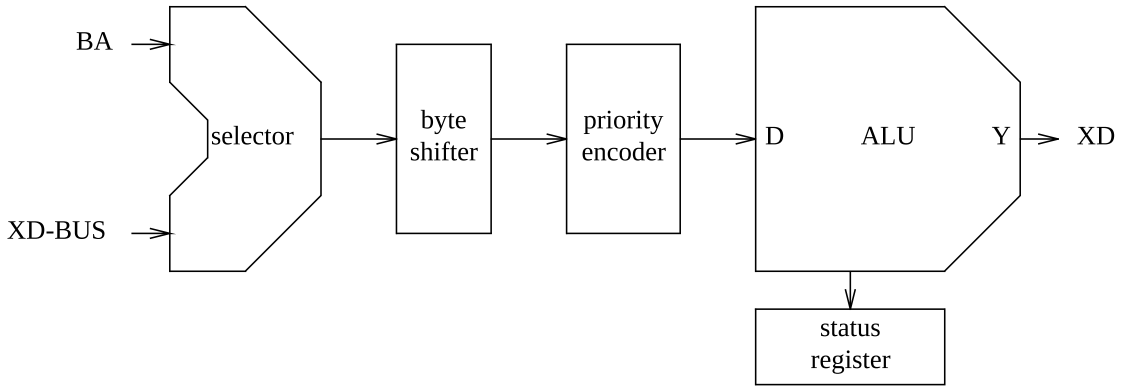

Figure 4.1 - Block diagram of complete ALU module

The ALU is made up from a number of components as follows:

#### 4.1.1. The Selector

Operates only on the bottom 12 bits of data, and selects these from one of two sources: the branch address (BA) from the micro-instruction or the XD-BUS. The top 20 bits always come from the XD-BUS, irrespective of which of these options are selected.

The branch address is shared by many parts of the processor. In the ALU component, it is used to provide an immediate value. Its other uses are: to provide a branch address for sequencer instructions such as CJP (jump) and to control the operation of the event logic.

#### 4.1.2. The Byte Shifter

The byte shifter is a simple barrel shifter that can rotate its 32 bit input by multiples of 8 bits. It is also able to force certain groups of 8 bits in the output to zero.

The byte shifter provides a number of rotate instructions RR[0-3] and also a number of rotate and mask instructions BYTE[0-3]. The rotate and mask instructions are provided to rotate the least significant byte from the input to one of the four byte positions in the output word - the other byte positions are set to 0. The byte ordering is little endian, with byte 0 being the least significant byte and byte 3 being the most significant byte.

The BYTE instructions are useful for reading several bytes into a 32 bit word register - by OR'ing the values into an initially zeroed register. The BYTE instructions are also useful for generating a selected range of 32 bit constants from the branch address (as may be chosen by the selector).

It is usually necessary to condition the branch address input before use in the ALU, because the value only appears on the lower 12 bits of the word - the other 20 bits always coming from the XD-BUS. An exception to this requirement is when the ALU is handling microcode addresses, in which only the lower 12 bits will have any significance - in this case, the value can be used directly in the ALU for address calculation without concern over the value of the upper 20 bits.

### 4.1.3. The Priority Encoder

The priority encoder operates only on the bottom 8 bits of data. It gives an output indicating the most significant 0 bit in the input. The top 24 bits are passed through unchanged. This output gives a bit number that is reversed, giving an output as follows:

| most significant 0 | output |
|---|---|
| bit7 | 0 |
| bit6 | 1 |
| bit5 | 2 |
| bit4 | 3 |
| bit3 | 4 |
| bit2 | 5 |
| bit1 | 6 |
| bit0 | 7 |
| no zeros | 7 |

The priority encoder can be selected to priority encode the bottom 8 bits of data or to pass all data through unchanged.

The priority encoder can be used for examining 8 bit status fields and determining if any of the bits are zero and if so the bit number of the most significant zero. Status fields of this type can be used to hold queues of outstanding actions.

### 4.1.4. The ALU component itself

The ALU is made from pair of Am29C101, 16 bit slices, giving a 32 bit ALU. The shift i/o pins of the two slices are connected in a circle, to enable data to be rotated in either direction.

The operation of the ALU component follows the normal rules for this device as described in the component data sheet [4].

### 4.1.5. The ALU carry input

The ALU carry input is derived directly from a bit in the microcode. It is not possible to select a value from the status register - this was not thought to be necessary as multi-length arithmetic operations are not likely to be needed for this application when using a 32 bit ALU.

The ALU carry input has two values:

| | |
|---|---|
| C | CIN = 1 |
| NC | CIN = 0 (default) |

It is important to note that the subtract instruction has an inverted carry input. This requires ‘NC’ to be selected when performing a standard ‘ADD’ operation and ‘C’ to be selected when performing a standard ‘SUBR’ or ‘SUBS’ operation.

### 4.1.6. The Status Register

The status register is a four bit register that may be updated at the end of any micro-cycle from the output status of the ALU. Three of the status bits correspond to status outputs from the ALU. The fourth status bit indicates if the bottom eight bits of the data (on the XD-BUS) contain a value that is greater than 240 - this is for use in page mode access to memory to give early warning of end-of-page.

The updating of the status register is controlled by the micro-code as follows:

| | |
|---|---|
| NOFLAG | don’t update status register (default) |
| FLAG | update status register |

The status bits themselves are referred to as: carry (C), zero (Z), negative (N) and end-of-page (EOP). The status register is not accessible via the data paths and may only be used to generate condition codes for conditional sequencer operations - such as conditional jump (CJP).

## 4.2. The XD-BUS

The XD-BUS is a 32 bit bus, acting as the main data highway within the processor.

There is one explicit source and one explicit destination for this bus during each micro-cycle. There are also a few other devices that can take a value from the XD-BUS during any micro-cycle irrespective of the current explicit destination.

Some combinations of source and destination are invalid, where this is limited by the hardware itself. In these cases, the address decoding logic will avoid any possible bus clashes. The main combination that is prohibited is the use of the transputer link adapter as source and destination - this is because the link adapter has only a single bidirectional port.

The XD-BUS is controlled by a pair of the fields in the micro- instruction as follows:

### 4.2.1. XD-BUS source

The XD-BUS can have six different sources.

| | |
|---|---|
| XSVPI | Video plane bus |
| XSRAM | Read Ram at address specified by Ram Address Register (RAR) |
| XSALU | ALU output port (default) |
| XSIDR | Transputer Link Adapter, Input Data Register |
| XSISR | Transputer Link Adapter, Input Status Register |
| XSOSR | Transputer Link Adapter, Output Status Register |

The Video plane bus, can provide a piece of data from a video plane memory card or other cards later to be plugged into this bus.

The RAM is the local memory for the processor. This is accessed via the RAM address register RAR, which can be loaded from the XD-BUS and also incremented and decremented - see later.

The ALU output port is the data from the ALU Y port.

There are three registers that may be read in the transputer link adapter. The one normally used will be the Input Data Register, the other two will only normally be used when polling.

### 4.2.2. XD-BUS destination

There are 6 selectable destinations for the XD-BUS, these are as follows:

| | |
|---|---|
| XDVPO | Video plane bus (default) |
| XDRAM | Write to RAM at address specified by RAR |
| XDVEC | The vector register |
| XDODR | Transputer link adapter output data register |
| XDISR | Transputer link adapter input status register |
| XDOSR | Transputer link adapter output status register |

The data on the XD-BUS is written out to the video plane bus by default. This may be used by devices on the video plane bus. The control of the video plane bus is specified in a separate section.

The Vector Register is a link between the ALU and the Sequencer. It provides a way in which addresses calculated in the ALU may be passed onto the sequencer to be used as branch addresses. The main use of this facility is to provide a form of ‘computed goto’ operation. This can be very useful in providing a fast form of ‘switch’ operation, to select a piece of code for execution without performing multiple tests.

The register normally accessed in the transputer link adapter is the output data register. The other two registers are provided to enable a program on start up, or at other times, to reset the transputer links and control the generation of events from transputer link i/o activity.

### 4.2.3. Other destinations for the XD-BUS

One destination of the XD-BUS has been covered elsewhere, this is the data input to the ALU module.

The other destination of the XD-BUS is the RAR (RAM address register), this will be covered in the next section.

### 4.2.4. The RAM module

The RAM module consists of an area of static RAM and the RAM address register (RAR). The RAR provides a 16 bit store address to the memory.

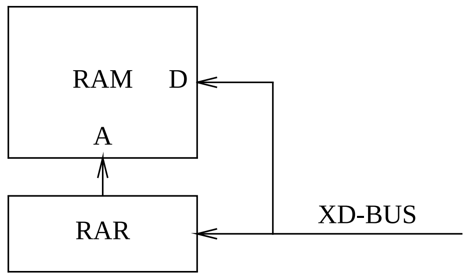

Figure 4.2 - The RAM module

The RAM consists of 32K by 32 bit word accessible memory (i.e., 128K bytes). This can only be accessed as words, not as bytes. The memory addresses are also word addresses - incrementing the address will point to the next word in memory.

The RAM can be read from or written to at the address specified by the RAR.

The RAR, consists of a loadable bidirectional counter, which is controlled by a field in the microcode. Several operations are possible:

| | |
|---|---|
| RARHD | Hold value, no operation |
| RARCLR | Load RAR with 0 |
| RARDN | Decrement RAR |
| RARLD | Load RAR from XD-BUS |
| RARUP | Increment RAR |

Although there is normally a two stage operation in accessing memory - load RAR, then access RAM - this is not so much of an overhead when accessing consecutive locations in memory as it is possible to autoincrement the RAR in the same cycle as the memory access. It is perfectly valid to perform both memory access and autoincrement in the same cycle as the RAR will not be updated until the end of the microcycle.

## 4.3. The Sequencer module

The sequencer module is based on the Am29C10A sequencer chip. As well as this, a number of extra functions have been added to enable the management and scheduling of a number of microcode tasks.

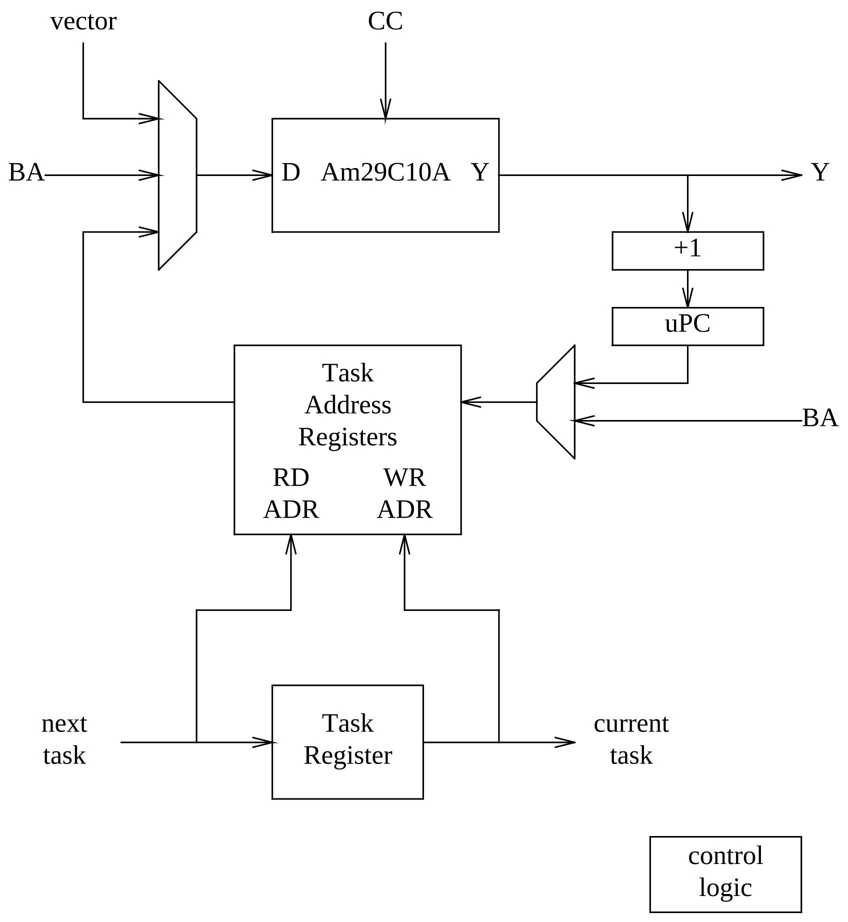

Figure 4.3 - Sequencer module

The Am29C10A provides most of the standard sequencer functions. This is controlled by the lower 4 bits of the 5 bit control field for the sequencer module. The Am29C10A can select one of three different sources to its ‘D’ input, these are selected from the following: BA from the control store (often referred to as PL for Pipeline), VEC from the Vector register and MAP from the task address register bank (in effect a MAPping RAM).

The instructions offered by the sequencer module consist of the standard 16 instructions provided by the Am29C10A and 4 extra instructions that relate to the scheduling and task registers.

An unusual feature of the sequencer module, is the ability to handle a number of microcode tasks and swap between them. There are up to 16 tasks in total, and the task number of the current task is held in the task register. There are also 16 task address registers, which contain the restart address for each of the 16 tasks. The task address registers take the place of the mapping prom that one would normally find on a conventional processor.

The sequencer module outputs the number of the current task for any other part of the system that wishes to know. It also has a ‘next task’ input from the event module, which gives the task number of the task to be executed next. The extra sequencer instructions handle the swapping between tasks. There is no forced scheduling between tasks, it is left to the programmer to de-schedule the current task or offer to do so. If rescheduling takes place then the address of the next task is fetched from the appropriate task address register - the continuation address for the current task may also be saved in the task address register for the current task. The task address registers are dual ported and may be read and written in a single microcycle. This allows the swapping between tasks, without any significant scheduling overhead.

The continuation address for a task may be either the address of the next instruction, or one specified by the programmer in the BA field. The choice to de-schedule or not can be made on the basis of whether the current task would be the next task to run if rescheduling takes place - meaning de-schedule if another higher priority task is waiting or the current task is not ready.

### 4.3.1. Sequencer Module Instructions

There are a total of twenty instructions for the sequencer module, fifteen of these are normal Am29C10A instructions and the other five are modified or extra instructions to manage the task scheduling.
The operations performed by the sequencer module are as follows:

| Instruction | | CC- == 1 | | | CC- == 0 | | | Task Logic | |
|---|---|---|---|---|---|---|---|---|---|
| | | stack | y | R | stack | y | R | TR | TAR[TR] |
| JZ | | clear | 0 | - | clear | 0 | - | - | - |
| CJS | | - | PC | - | push | BA | - | - | - |
| JMAP | | - | TAR[NT] | - | - | TAR[NT] | - | - | - |
| CJP | | - | PC | - | - | BA | - | - | - |
| PUSH | | push | PC | - | push | PC | - | - | - |
| JSRP | | push | R | - | push | BA | - | - | - |
| CJV | | - | PC | - | - | BA | - | - | - |
| JRP | | - | R | - | - | D | - | - | - |
| RFCT | R!=0 | - | F | dec | - | F | dec | - | - |
| | R==0 | pop | PC | - | pop | PC | - | - | - |
| RPCT | R!=0 | - | D | dec | - | D | dec | - | - |
| | R==0 | - | PC | - | - | PC | - | - | - |
| CRTN | | - | PC | - | pop | F | - | - | - |
| CJPP | | - | PC | - | pop | BA | - | - | - |
| LDCT | | - | PC | BA | - | PC | BA | - | - |
| LOOP | | - | F | - | pop | PC | - | - | - |
| CONT | | - | PC | - | - | PC | - | - | - |
| TWB | R!=0 | - | F | dec | pop | PC | dec | - | - |
| | R==0 | pop | D | - | pop | PC | - | - | - |
| NEXT | | - | BA | - | - | TAR[NT] | - | NT | PC |
| WAIT | | - | PC | - | - | TAR[NT] | - | NT | PC |
| TRCLR | | - | PC | BA | - | PC | BA | clear | - |
| TRLDI | | - | PC | - | - | PC | - | inc | BA |

### 4.3.2. Standard Sequencer Operations

This document will not attempt to explain the operation of the fifteen standard sequencer instructions. For details of these instructions, the reader should consult the Am29C10A chip specification [5].

### 4.3.3. Task and Scheduling Instructions

There are a number of operations that need to be performed to handle the scheduling of microcode tasks. In this implementation there are up to sixteen tasks, each with a task address register. The task address registers are stored in an area of dual port memory that may be accessed via a read and a write address. The read address is obtained from the task number of the next task to be scheduled and the write address is obtained from the task number of the current task. At initialisation, the task address registers will need to be loaded with the start address of the tasks they refer to, this is done using the TRCLR and TRLDI instructions.

When the microprogram is executing, the current task can offer to de-schedule. If rescheduling takes place, then a restart address for the current task can be saved in the current task address register. The start address for the new task to be executed comes from the task address register for that task. Because of the dual port task address register memory, these two operations can both happen in a single micro-cycle. When a new task is started, the task register will be updated to show the task number for that task as the current active task. The offers to reschedule are performed using the: JMAP, NEXT and WAIT instructions.

### 4.3.4. TRCLR

The TRCLR instruction is "Task Register Clear". This simply loads the task register with 0, setting 0 as the current task number. ( TR = 0 )

This instruction also loads the R register of the Am29C10A from the BA, as a side effect.

### 4.3.5. TRLDI

The TRLDI instruction is "Load current task address register from BA and then increment the task register". ( TAR[TR++] = BA )

This instruction is used to load task start addresses into the task address registers. The method of loading the task addresses at initialisation is expected to be as follows:

```
TRCLR
TRLDI BA=task0
TRLDI BA=task1
TRLDI BA=task2
  .
  .
  .
TRLDI BA=task14
TRLDI BA=task15
```

These instructions may be used while the microcode tasks are running, but the programmer needs to take care that the task register (TR) contains a correct value if the current task intends to de-schedule and save state.

### 4.3.6. JMAP

The JMAP instruction is very similar to the standard use of this instruction with the Am29C10A. In this case however, the input to the Am29C10A will come from the task address register for the next task - rather than a mapping prom.

The JMAP instruction is used to reschedule unconditionally, without saving a restart address for the current task. This can be used to de-schedule from tasks that will always be entered from the same address - such as a short piece of code that is executed from start to completion.

This instruction should be used after initialisation, to start the scheduling process, as it will not disturb the task address registers.

### 4.3.7. NEXT

The NEXT instruction is used to reschedule conditionally and save state. The NEXT instruction forces a condition code select of NOTRDY.

The NEXT instruction is similar to the CJP operation. It takes a branch address of where to execute next. The difference is that this can allow the task to be de-scheduled under certain circumstances. If the task is in state READY and is the highest priority task that can be scheduled, then the branch will be taken. If the task is not in state READY or there is a higher priority task that can be scheduled, then the current task will be de-scheduled and rescheduling will take place. The branch address will be saved in the current task address register.

If a task using the NEXT instruction is de-scheduled, then it will restart later at the address given by the branch address.

It is important to ensure that any task offering to de- schedule in this way, has a chance of being rescheduled again later. The task should either be in state READY or expect to be so at a later time. It should also be in state RUN or have an expectation of another task putting it into this state.

### 4.3.8. WAIT

The WAIT instruction is another way to reschedule conditionally and save state.

The WAIT instruction is like a CONT operation. The WAIT will specify the next instruction as the place to execute next. The WAIT instruction will however allow the task to be de- schedule under certain circumstances. If the task is in state READY and is the highest priority task that can be scheduled, then the task will continue executing. If the task is not in state READY or there is a higher priority task that can be scheduled, then the current task will be de-scheduled and rescheduling will take place. The next address will be saved in the current task address register.

If a task using the WAIT instruction is de-scheduled, then it will restart later at the next instruction.

Again, it is important that a task offering to de-schedule in this way has a chance of being rescheduled again later.

The WAIT operation is useful on sequences of i/o operations, such as reading a number of bytes from the transputer link adapter. If four bytes are being read from the link adapter into a 32 bit register, then the first three reads can have a WAIT instruction appended to the line of microcode to cause the task to be de-scheduled between i/o operations. This will cause the task to wake up for one microinstruction for each i/o activity.

## 4.4. Event Control and Condition Code select

The event control logic is responsible for selecting tasks for execution. In each microcycle, the event control logic generates the task number of the task that it considers should be executed next. This next task (NT) value may be used by the sequencer module for scheduling. The event control logic also compares the next task number with the current task number to determine if the current task is the one that should currently be running.

The event control logic has a number of event inputs from other parts of the system each of which indicate whether the task is READY to execute or should WAIT. These inputs are similar to interrupt or attention signals. The event logic also has an event enable register, that determines which of the tasks are currently active (state RUN) or inactive (state HALT). The only tasks that are considered for execution are those ones that are both in state RUN and have an event input of READY.

The event inputs are derived from various pieces of hardware in the system and the task state comes from the event enable register. The event enable register is controlled by micro- code. So any task may change the state of any other task (including itself) between states RUN and HALT. This task control has to be used with care to ensure tasks are not put into a HALT state from which they never leave. The normal mode of operation is for a task to change its own state into state HALT when it has no more work to do and for other tasks to change its state to RUN when they want it to do some work for them.

### 4.4.1. The event enable register

The event enable register is a 16 bit register that is bit addressable. The contents of the event enable register may only be changed by using an event command:

| | |
|---|---|
| EVNOP | No event command (default) |
| EVCMD | Event command |

When issuing an event command, the event enable register uses the data from the branch address (BA). Rather than loading a value into the event enable register (EER), the branch address is used as a pair of bit addresses to indicate which bit is to set and/or cleared. Bits BA[4..0] defines the number of the bit to set (0x1n where n is bit number) or no operation (0). Bits BA[9..5] define the number of the bit to clear (0x1n where n is bit number) or no operation (0).

Therefore in one microinstruction one EER bit may be set and one EER bit may be cleared. It needs to be remembered however, that these fields are shared with the branch address (BA) field.

### 4.4.2. Priority encoding

The event inputs are all gated with their corresponding event enable bits and the outputs fed into a priority encoder. The priority encoder gives the bit number of the most significant 1 in its 16 bit input. If none of the 16 inputs is 1 then the priority encoder gives a default output of 0.

The output of the priority encoder is used as a next task (NT) number. This next task number is compared with the current task number to give an NT == TR output and an NT > TR output - where TR is the value from the task register giving the number of the current task. These logical outputs are used for scheduling and condition codes where NOTRDY = !(NT == TR) and HIPRI = (NT > TR).

### 4.4.3. Condition code select

The condition code select logic has six different inputs: the four ALU status bits (Z,C,N,EOP) and the two task priority bits (NOTRDY and HIPRI) as defined above. At present, only five conditions may be selected by the condition code select logic, these are as follows:

| | |
|---|---|
| TRUE | always true cc=1 |
| FALSE | always false cc=0 |
| Z | true if Z bit set cc=Z |
| NOTRDY | true if current task and next task differ cc=!(NT == TR) |
| HIPRI | true if a high(er) priority task is waiting cc=(NT > TR) |

The last two look similar, but are actually for quite different purposes. The NOTRDY test allows a task to check if it would be de-scheduled if it executed a WAIT or NEXT operation. The HIPRI test just checks to see if a task is waiting to execute which is of a higher priority - this is independent of whether the current task is capable of being rescheduled at present.

There are three more condition codes that may be selected. The function of these have yet to be defined, but two of these will be used for particular task specific functions for breaking out of tight inner loops.

Note: the sequencer uses a cc- input rather than a cc input, so the actual boolean output value will be inverted.

## 4.5. The Video Plane Bus Interface

The Video Plane Bus consists of a relatively slow synchronous bus. All data and addresses are transferred over a 32 bit data bus AD[0..31]. The current function of the bus is defined by a 16 bit command VC[0..15], which is used to control the video planes and a 4 bit command SF[0..3] which is used to control everything else. The current task is shown by BCT[0..3], which just gives the task number. External events are indicated by BEV[4..15]- which are used to tell the processor when a task is to be run.

The video plane command also controls the timing of both the internal processor and the video plane bus.

### 4.5.1. Memory Cycle type and length

This field determines the type of memory cycle performed by the video plane memory and also the length of the micro-cycle.

At present, only single word access to memory is fully supported.

| | |
|---|---|
| T250 | 250ns microcycle (default) |
| T300 | 300ns microcycle |

The 250ns cycle is used for all microcycles except those performing scheduling operations. The 300ns cycle should be used for any line of microcode containing one the following sequencer instructions: JMAP, NEXT, WAIT, TRCLR or TRLDI.

Later versions of this system will support page mode access to memory. This will allow two words of data to be transferred in one 300ns microcycle and will be used mainly for moving data from the fifo to the video plane memory during video input.

## 5. Results and conclusions.

### 5.1. The Task Scheduling system.

This control processor has been successfully built and used to control the video server. The hardware scheduling system appears to work correctly and enables a number of tasks of varying priority to be run on the same system.

The only difficulty in programming is to ensure that the lower priority tasks do not hog the processor for too long. There is a tendency to spawn off tasks that will execute for long periods of time - and some cases as a base level operation that is lower priority than any other functions occurring on the system.

When writing these low priority tasks, it is important to ensure that they offer to deschedule at regular intervals, even if they have no wish to do so. This sounds to be rather difficult to manage, but in practice can be incorporated into simple loop constructs by using the deschedulable jump instruction (NEXT). Alternatively, if the code is operating in a tight loop, then the code can check to see if any higher priority event is waiting by testing one of the condition code flags. If a higher priority event is waiting, then the current task can tidy up, prepare for re-entry and then deschedule itself.

### 5.2. Method of use.

The remote transputer system manages the operation of the video server. A client server interface exists between the two, with the transputer system sending commands to the video server for execution. These commands allow data to be read or written in the video controller or video plane memory and for video input and output to be started and stopped.

All the data for the control structures are created by the transputer and down loaded into the video controllers memory. This is done each time the window was resized - to set up the control data structures. Also, moving or resizing the windows causes the clipping information to be changed and requires the video attribute plane to be updated.

Originally, the attribute plane was updated by a word-fill command sent over the transputer link. This ran a simple piece of microcode that scheduled a single word write at a time. However, the pixels themselves are layed out on a byte boundary. The short term solution to this was for the transputer to request the first and last words of a line of pixels to be fetched, they were then modified by the transputer and sent back. This whole procedure was rather slow, so a number of measurements were taken.

| operation | time(ms) |
|---|---|
| read word | 30.9 |
| write word | 35.9 |
| word fill | 48.2 + 1.55/word |

The time for the per word fill was exactly as expected, but the overheads for each command were rather surprising. This problem was investigated and the results explained in the next section.

From the observations above, it was clear that to achieve good performance it was important to avoid sending too much data or too many commands over the transputer link. In the first implementation, a single fill operation required 5 separate commands to be sent over the link: one word fill command and a read and write operation for each end - assuming the fill was not word aligned. To improve the performance, the amount of communication between the transputer and video server needed to be reduced.

The original fill algorithm was replaced by a new piece of microcode that performed a byte fill operation and achieved a greater performance due to the use of burst transfers of data to the attribute plane. This new piece of microcode was quite large and took some time to write and debug. However this gave an improved per pixel performance and also reduced the overheads:

| operation | time(ms) |
|---|---|
| byte fill | 46.0 + 0.126/byte |

It appears that the transputer link in this system should be treated more as a standard network connection to the transputer rather than a high speed data highway. Care needs to be given to identify the types of client server operation performed over this link as evidently in many of the cases used in this system, the communication overheads are significantly greater than the time taken to perform the processing required.

## 5.3. The Transputer Link.

As mentioned above, it was discovered that the transputer link did not give a very high data rate between this control processor and a transputer. There were many reasons for this low performance, mainly to do with the way the link was handled by the video control processor.

The transputer link was connected to a transputer link adaptor [6] at the video control processor end. This gives a parallel interface to the transputer link and a pair of interrupt outputs to indicate ‘output buffer empty’ and ‘input buffer full’. These signals, after conditioning to the time system of the video control processor were used as event signals.

The data sent over the transputer link is sent a byte at a time preceded by a 2 bit header and followed by a single stop bit. When the data has been received by the transputer, a 2 bit acknowledgement is sent back that will initiate transmission of the next byte. With the T800 transputer, this scheme is modified [7] such that the receiving transputer sends back the acknowledgement immediately the header of the data is received, this gives a certain amount of pipelining in that the next byte can be made ready to be sent immediately the last byte has been sent.

The modified transmission algorithm does not appear to be implemented in the link adaptor, which will only return its acknowledge after the data has actually been read from the input buffer. This means that the control processor needs to read the data before the acknowledgement is sent back and the next byte transmitted. The time taken to respond to the event and to read the data from the input port is therefore in addition to the normal transmission time.

## 5.4. Future Work

The video control processor has been quite successful. It performs the functions required of it - that of handling several real time event driven tasks and can also perform a certain amount of main program type of functions.

The hardware is quite stable. There is no immediate reason to make any significant changes. It would however be interesting to make the necessary upgrades to allow the processor to drive page mode access to the video frame store memory - this would reduce the percentage of the processor time spent performing the video input operations. This is a trivial hardware modification, but would require rewriting some of the microcode so as to exploit this mode of memory access.

What would be interesting is to move some of the low level store management functions from the transputer and into microcode. This would need careful consideration as the production of microcode is not undertaken lightly as it can be time consuming to develop and debug. However, a number of simple functions to separate the transputer from the video server hardware would be advantageous. What would be needed is really a number of basic functions that allow the transputer to treat the video server more as a series of objects on which it can instruct operations to be performed and to hide the internal operation of the video controller and its memory system from the transputer.

## References

[1] TMS4461 262,144-Bit Multiport Video RAM. Advance Information. Texas Instruments. December 1986.

[2] The V-plane. Palantir report No.5. G.E.W. Tripp. UKC Computing Laboratory, Canterbury Kent. July 1989.

[3] M68000 16/32 Bit Microprocessor. Programmers Reference Manual. Motorola Semiconductors 1984.

[4] Am29C101 16-Bit CMOS Microprocessor Slice. Data Sheet. Advanced Micro Devices. April 1987.

[5] Am29C10A CMOS Microprogram Controller. Preliminary Data Sheet. Advanced Micro Devices. December 1985.

[6] IMS C012 link adaptor. Engineering Data. INMOS. The Transputer Data Book, 2nd Edition 1989.

[7] Transputer Applications Notebook. Link operation p30-31. INMOS. June 1989.