DMA Controller

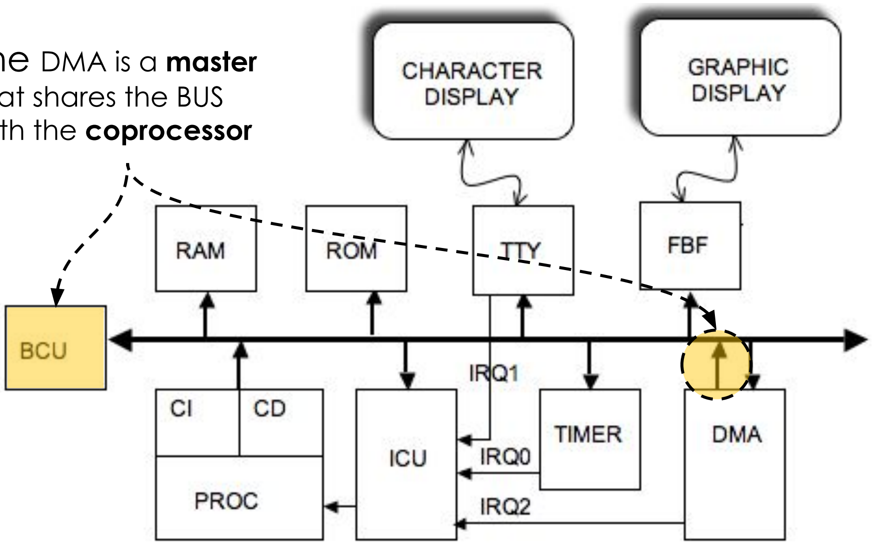
MU4IN106 Multi

Platform with DMA and FBF

The DMA is a peripheral capable of making direct accesses to the memory. It acts under the orders of the program as a co-processor.

The FBF is a graphic screen controller

The DMA is a **master** that shares the BUS with the **coprocessor**



The FBF (Frame BuFfer) controls a graphic display
- It is an address segment in the proc address space.
- This segment represents a memory of pixels
- image 128x128 → 16kiB

The DMA performs data **transfers** in bursts
Ordering:
- source address
- destination address
- length in bytes

Signals termination by **IRQ**

Plan

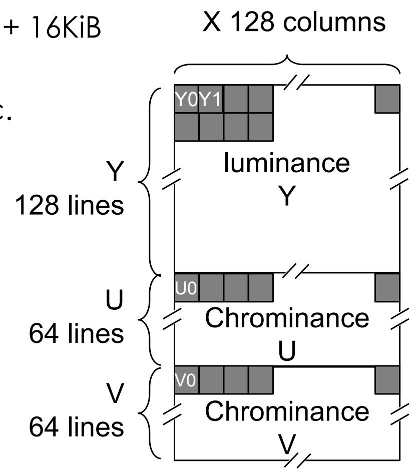
- Presentation of the new hardware components.
- Presentation of the parallel use of the processor and the DMA.

Hardware Components

FB device : Frame Buffer

It is a video controller, here it is a memory representing the YUV422 coded image

- On the platform the image is 128 x 128 pixels.
- Each pixel occupies two bytes:
  1 for luminance (light intensity) Y and 1 for chrominance U and V.
- The image therefore occupies 128 x 128 x 2 = 32KiB = 16KiB + 16KiB
- The first byte contains the luminance of the pixel (0,0) [Y0] placed at the top left, the second is the pixel (1,0) [Y1], etc. until pixel (127,0) then byte 128 contains pixel (0,1), etc.
- The color tables are located just after with the pixels arranged in the same order.
  The first byte of the first table contains the chrominance U of pixels Y0 and Y1.
  The first byte of the second table contains the chrominance V of pixels Y0 and Y1.



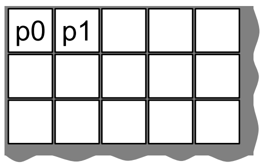
FBF device : Frame Buffer

The Frame Buffer component is a **target** device.

It has no addressable register in this version. It is a memory accessible in writing and in reading. The values that are written are sent to the screen.

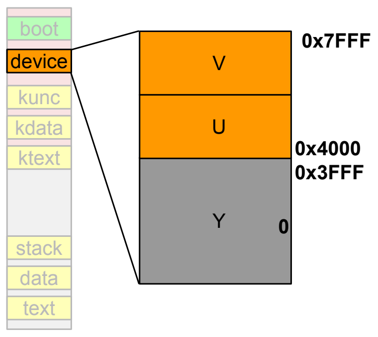
For each pixel, the screen recalculates the Red, Green and Blue components

For the pixel p0
r = (y0 - 0.00004*u0 + 1.140*v0)
v = (y0 - 0.395*u0 - 0.581*v0)
b = (y0 + 2.032*u0 - 0.0005*v0)

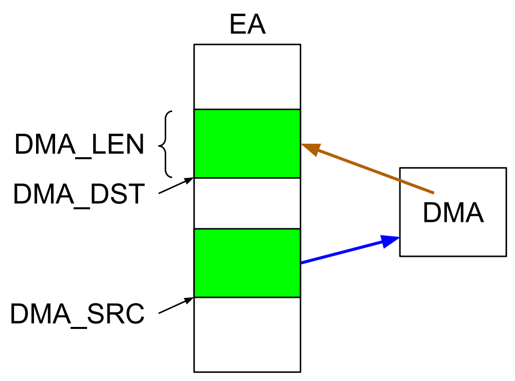


For the pixel p1
r = (y1 - 0.00004*u0 + 1.140*v0)
v = (y1 - 0.395*u0 - 0.581*v0) p0 and p1 share the
b = (y1 + 2.032*u0 - 0.0005*v0) same chrominance

etc.



DMA device: Direct Memory Access



The component receives transfer orders

If it is the user who makes the request, it goes through a system call, his request is checked in order to forbid him to write in the address segment reserved for the system (> 0x80000000)

The system must also manage the fact that the DMA *channel* is free. Indeed, this channel may already be used by another program.

If the DMA channel is already in use, the new requester must be put on hold, until the current copy is terminated.

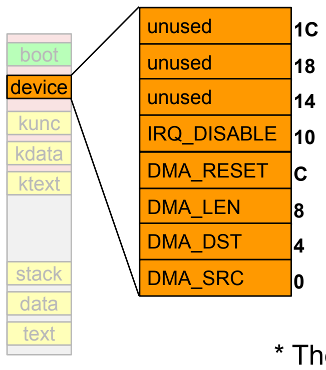
DMA device: Direct Memory Access

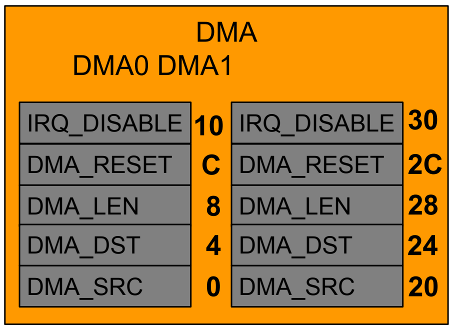
The DMA performs a memory copy
from the DMA_SRC address to the DMA_DST address of DMA_LEN bytes.
In order, we start by writing the addresses SRC, DST and IRQ_DISABLE (if necessary),
then we write LEN, which causes the copy to be started by the DMA

- DMA_IRQ_DISABLE (read/write) hiding the IRQ line
- DMA_RESET (write only) IRQ line acknowledgement
- DMA_LEN (write/read) size in bytes to move
- DMA_DST (write only) destination address
- DMA_SRC (write only) source address

At the end of the operation, the DMA raises an interrupt and LEN contains the number of unwritten bytes.

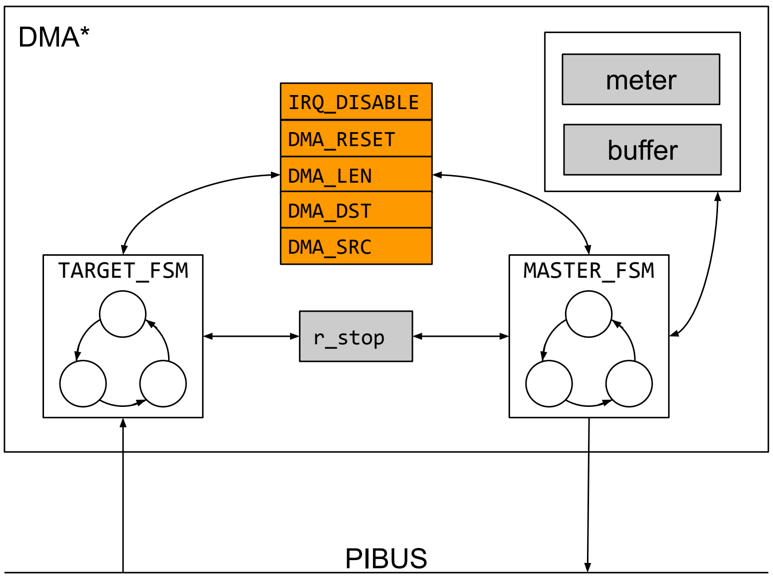
If it is different from 0, it means that there is an error.





* The drawn component contains 2 DMA, in the platform, we put one per MIPS

Internal architecture of the DMA



** This drawing does not contain all the DMA registers*

There are two separate FSM

TARGET_FSM
    that responds to orders
    of the processor
MASTER_FSM
    which executes the commands

The DMA must be able to accept commands from the processor while executing its own commands

But these automata must synchronize themselves ⇒ the r_stop flip-flop

1 : `MASTER_FSM`	is stopped in waiting for an order
0 : `MASTER_FSM`	executes a command

In TME, you will watch when **`r_stop`** changes state.

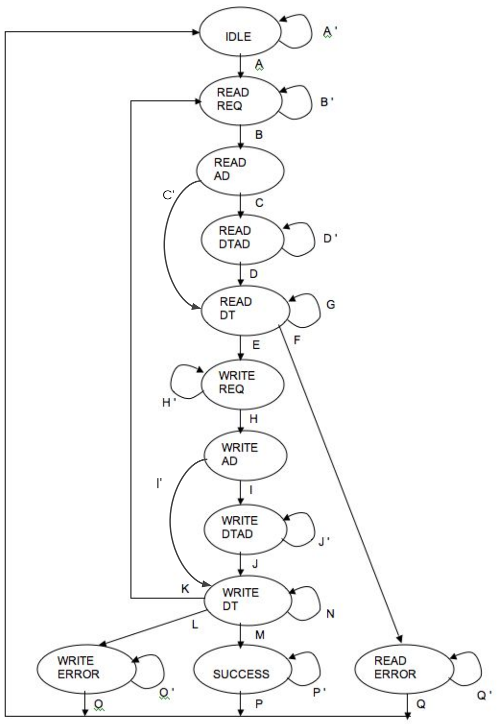
`MASTER_FSM` state transition graph

The DMA makes several reads to fill a buffer, then as many writes (read/write bursts) in order to save the bus ownership request and the "dead" cycle present between exchanges.

In TME, you will read and analyze the behavior of the `MASTER_FSM` controller

Let's start here * :

- The `MASTER_FSM` automaton is first waiting for a command to be executed.
  `A = ?`
- Then, it must ask for the ownership of the BUS
  `B = ?`
- Then, it sends the address to the source address
  `C = ?`



* these are not the right signals

`MASTER_FSM` state transition graph

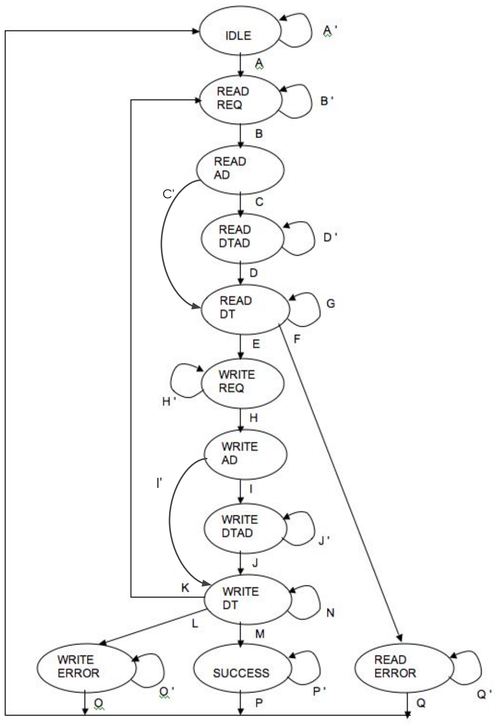
The DMA makes several reads to fill a buffer, then as many writes (read/write bursts) in order to save the bus ownership request and the "dead" cycle present between exchanges.

In TME, you will read and analyze the behavior of the `MASTER_FSM` controller

Let's start here * :

- The `MASTER_FSM` automaton is first waiting for a command to be executed.
  `A = *not* stop`
- Then, it must ask for the ownership of the `BUS`
  `B = grant`
- Then, it sends the source address
  `C = not last`

*these are not the right signals



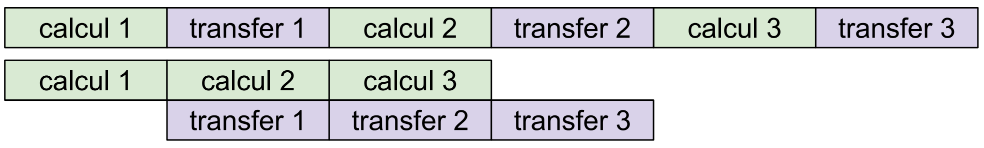
Software Usage

Software pipeline concept

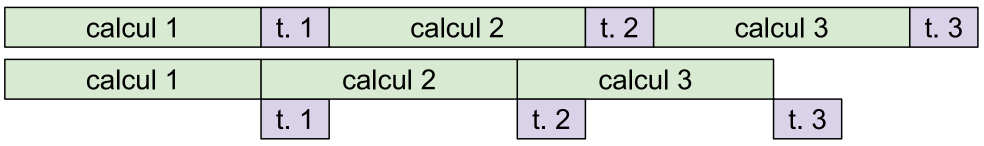
- Let's suppose that we want to perform a treatment on a data flow
  - image1 production by MIPS
  - transfer image 1 to screen
  - image2 production by MIPS
  - transfer image 2 to screen
  - image3 production by MIPS
  - transfer image 3 to screen
  - [...]
- The DMA is a co-processor of the MIPS, it can work in parallel
  - image1 production by MIPS
  - produce image2 by MIPS & transfer image 1 to the screen
  - produce image3 by MIPS & transfer image 2 to the screen
  - produce image4 by MIPS & transfer image 3 to the screen
  - [...]
- The DMA does not check the source and destination addresses, it is necessary that the kernel does these verifications to avoid reading or writing in the kernel code or data

Software pipeline concept

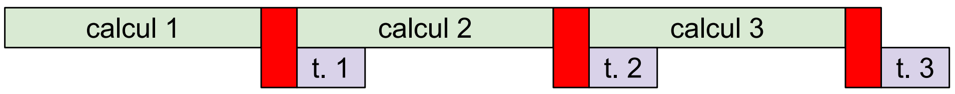
- We can divide by 2 the apparent processing time for an image



- but if the duration of the treatments (MIPS and DMA) are very different, it is less



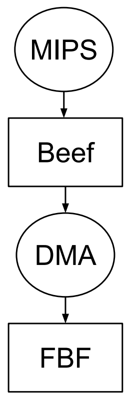
- and if there are data copies and synchronizations, the gain can be null...



→ You need to know the relative duration of the steps and assess the extra cost (the red zone)

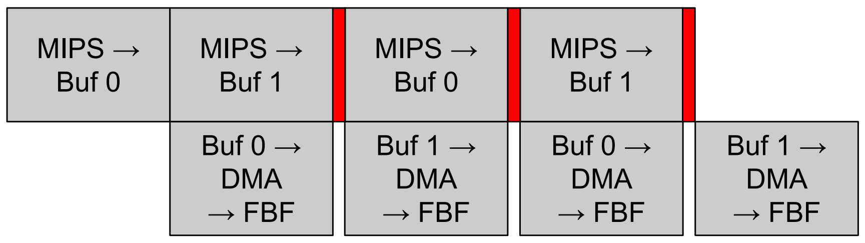
Sequential operation

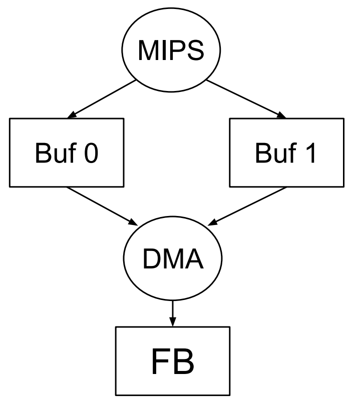
- The MIPS produced in a buffer
- then order the DMA
- The MIPS must wait for the DMA to finish before it can reuse the buffer
- MIPS and DMA must be synchronized:
  - or the system call to write to the FBF using the DMA is blocking
    → fb_sync_write()
  - or the system call to write to the FBF is not blocking
    → fb_write()
    but then we need a way to know that the transfer is finished
    by adding a blocking system call (it could have been not
    blocking but just a state and we would loop to wait)
    → fb_completed()



Pipeline operation

- To parallelize, 2 buffers must be used alternately





let b be the buffer number, initialized to 0

1. startup: MIPS produced in buf b
2. steady state : MIPS command DMA buf b
   b = 1 - b
   MIPS produces in buf b
   MIPS waits for the DMA to finish

Question: do the buffers have to be in an unhidden region to avoid cache consistency problems?

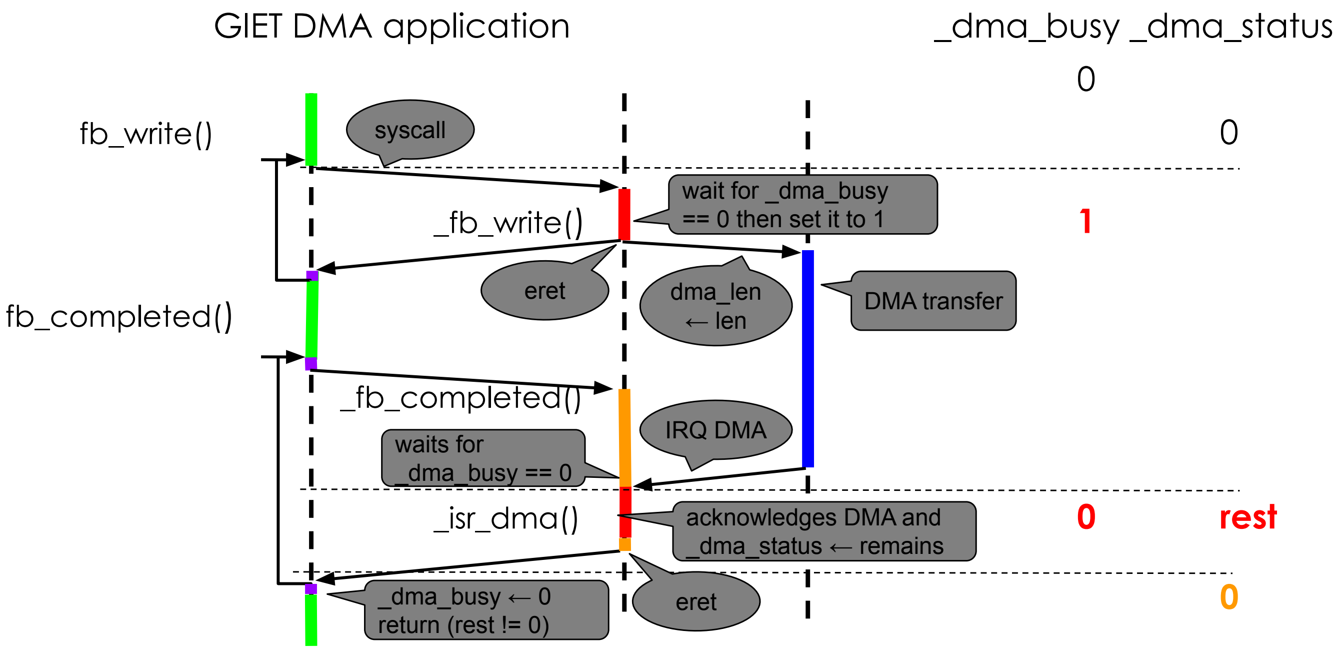
Application and device synchronization

The DMA device must start when the MIPS has finished computing

The system uses a global variable indicating the state of the DMA: _dma_busy

- `_dma_busy` is 0 at startup
- the application takes the DMA (with `_dma_lock`), makes its command and then resumes its work.
- the DMA starts to make its transfer
- when the DMA finishes, it raises an IRQ
- which causes the execution `_isr_dma`() whose work will be
  - to read the result status
  - to pay for the interruption
  - set to 0 `_dma_busy`
- When the application needs the DMA, it must set `_dma_busy to` 0 in an active wait (in the kernel)

Synchronization: Application - Device



We have seen

- That a DMA capable operator is a co-processor
  that can perform operations in parallel with the MIPS
- That this operator is both a target and an initiator.
- That its use requires a synchronization
  to guarantee the good use of the shared buffers.
- That the DMA has all the rights and that
  it is the kernel that must check the user's commands.

In TME

- After the study of the DMA master controller,
- you will experience the parallel use of MIPS and DMA
  - first without DMA, the processor does everything
  - then a DMA and 1 single MIPS
  - Then with several MIPS because to gain in performance