WAKE UP YOUR WAY ALARM CLOCK

by
Eleanor Foltz and Jon Spurlock

6.111 Final Project
May 12, 2005

ABSTRACT JS

The average alarm clock provides a poor wake-up experience due to its abrupt nature and the vice that is the snooze button. In this document, we describe the design and implementation of an alarm clock that uses a combination of light and sound over an extended period of time. The functions of our alarm clock are as follows: a standard buzzer, a gradually brightening light, implemented via a duty cycle controlled square wave, audio playback from mp3 files using a mp3 decoder chip, and voice memo recording and playback. The alarm clock has a programmable 'main' alarm time, and each function can be independently started at a positive or negative programmable time offset from the main alarm time. The alarm clock makes use of an LED display to provide a visual indication of time and to allow for configuration of alarm settings.

INTRO AND USER INTERFACE JS

The Wake Up Your Way Alarm Clock (WUYWAC) was designed to alleviate the harsh and abrupt experience associated with standard alarm clocks by adding multiple audio and visual cues that can gradually "build in" over an extended period of time based on some primary alarm time. These additional functions include a light of gradually increasing intensity, mp3 music, and a recordable voice memo in addition to the standard buzzer. Each of these four functions can be programmed to turn on independently at a user specified time offset from the main alarm time. So, for example, a user might set the primary alarm time for 7am, but program the light to begin turning on at 6:40am, the music at 6:50am, a voice memo reminding him or her of a 9am meeting to play at 7am, and finally a "just in case" buzzer to go off at 5 after 7am. Control and programmability of the WUYWAC is accomplished solely through the push buttons and toggle switches provided by the new 6.111 lab kit. Toggle switches were used to turn the alarm on and off and to set the alarm time, clock time, and four function alarm time offsets. Push buttons were used to reset the system and to provide increment or decrement inputs for adjusting times (via the D-pad, an homage to the classic video game controller directional pad which four of the labkit buttons reminded us of).

ARCHITECTURAL OVERVIEW EF

This project was implemented on the new labkits provided by the 6.111 staff. These kits contain a 6 million gate FPGA which is programmed using Xilinx ISE software. Specific credit is due to Nathan Ickes for his code (display.v and the labkit.v template). Most of the functionality was contained within the FPGA, but addition external circuitry included A/D and D/A converters, an op-amp amplifier, LEDs, and an MP3 encoder chip (STA013). Two main modules discussed are the MENU and ALARM modules. The MENU module keeps track of the delta times associated with each alarm function, which are time offsets from the main alarm time. The ALARM module triggers each of these functions if the alarm is on and turns them off when the I'M UP button is pressed.

The design consists of several modules all contained within a top level labkit module. All of the user inputs were synchronized, delayed two cycles of the 27MHz clock. Most were synchronized directly at this level, but the up, down, right, and left buttons were synchronized within a module that created a variable called D_pad for use within other modules. The menu module was instantiated within labkit.v. It took as inputs the four switches and the D_pad. It output all four of the delta times to the alarm module. It also output an alarm function number and a single delta time for that alarm function to the displaymsg module. The displaymsg module instantiated at this level outputs the large dots variable sent to the display module, also at this level. It uses the set alarm, set clock, and set feature switches from the labkit and the menu module to know which time to display. The alarm module containing the control signals is instantiated as well. It ouputs separate start and stop signals to the four alarm functions, each of which has a module at the labkit level. The clock module makes available both the current time and a 1Hz clock. These modules are described in more depth below.

CLOCK EF

The clock module uses the internal labkit signal at 27MHz to create a 1Hz signal. It also outputs an 18 bit clock time comprised of 1 unused bit (MSB) which is reserved for a future implementation of a military vs standard clock indicator, 5 bits for hours, 6 bits for minutes, and 6 bits for seconds (LSB). A toggle switch on the labkit allows changing the current clock time. Pushing the up or down buttons will create a single pulse in the D_pad variable. If the setclock switch is on, the clocktimefsm module instantiated with the clock module will adjust the clock time up (later) or down (earlier). A state diagram for this module is included at the end as Figure 3.

Since we decided to represent time in hours, minutes, and seconds, current_time + 1second is not necessarily equal to the binary representation of current_time + 00000000000000001. Jon wrote a time_alu which allows the addition of two time values. This was particularly useful with the controlling alarm module, but was also used in the clock module to add 1 second to the current time every one second.

The user has an interface to this module that changes the speed of the clock. In student mode, the speed switch pushes the clock faster. In professor mode, time passes at one second per second, because professors have a much better grasp of

reality than students for whom time passes all too quickly. (Seniors EF and JS are still quite happy to be finished!) A functional diagram is included at the end as Figure 2.

ALARM MODULE JS

The alarm module handles two main tasks: keeping track of a programmable alarm time and starting the four alarm functions at the correct time offsets from the main alarm time. The alarm module is broken down into a time submodule, main alarm submodule, and four function activation submodules that start or stop the buzzer, light, music, and voice memo alarm functions. A functional diagram is included at the end as Figure 4.

Alarm Time Submodule

The alarm time submodule consists of one simple state machine. On each positive clock edge, it checks to see if the 'alarm set' input is high. If not, the alarm time stays constant. If so, the submodule checks the D-pad input. If it detects that the up or down push button has been pressed (i.e. the corresponding D-pad bit is high), it changes the alarm to one minute later or sooner, respectively.

Alarm Main Submodule

The alarm main submodule takes as inputs the clock time, main alarm time, alarm on/off status, and the time offsets for each alarm function. If the alarm status is 'off', the alarm main module stays idle. If the alarm status is 'on', the submodule monitors the clock time and compares it to the main alarm time. As shown in the state diagram of Figure 5, when the clock time is less than the alarm time, the submodule enters a trigger wait state. Once the clock time is within half an hour (and still less than) the main alarm time, the main submodule starts the four function activation submodules. It also provides each function activation submodule with it's specific function alarm time calculated by adding the respective time offset to the main alarm time. Once each of these submodules indicate that they have been started (by raising a busy signal), the alarm main submodule waits for either the alarm to be turned off or for all four function activation submodules to end on their own (indicated by the busy signal going down). Finally, the main submodule waits for the clock to increase above the main alarm time before reentering a state where it waits for the clock time to return to being below and within a half hour of the main alarm time.

Alarm Function Activation Submodules

The alarm function activation submodules work similarly to the main submodule. Each of the function activation submodules stays idle until started by the main submodule. Once started, they confirm to the main submodule that they have been started, monitor the clock time, and compare it to their respective function alarm time, which is also provided by the main submodule. When the clock time is greater than a function alarm time, the function activation submodule starts its respective function module by raising a 'start' signal. Once the function module indicates that it has started, and then finished, the function activation submodule indicates to the main alarm module that it is done. If the main submodule stops the function activation submodules while their function modules are busy, they will stop their function modules by raising a 'stop' signal.

In addition to start and stop signals, the light function activation submodule also provides the light module with two signals to control the light intensity. One signal is the main alarm time plus five minutes minus the light function alarm time. The second signal is the current clock time minus the light function alarm time. Both signals are six bit signals corresponding to the minutes of the resulting time arithmetic operations. The first six bit signal will be larger than the second so long as the clock time is less than five minutes past the main alarm time. Once this condition no longer holds, the light function activation submodule sets both signals to be equal.

MENU EF

The main goal of the menu module is keep track of the four delta times for the four alarm functions. Additionally, the menu module sends some information directly to the displaymsg module. The menu module instantiates its submodules and contains a case statement. There are no state machines at the menu level. Four identical, parallel FSMs are instantiated in the smallmenufsms which control the editing of the four times. The user indicates he wants to change an alarm function time by selecting one of 4 switches on the labkit. Each statemachine can be hard-programmed with a maximum and minimum value for the time that that function will activate. If more than one switch is selected, the internal module menuchoose will make the choice for the user so that only one delta time is adjusted by any push of the D_pad buttons. Both the alarm function number and the delta time being adjusted are sent to the displaymsg module.

The delta times are in sign magnitude format to ease both display functionality and the adding and subtracting within the alarm module. Figure 6 and figure 7 are the block diagram of the menu module and the state diagram of the FSM.

DISPLAY EF

The display consists of two instantiated modules at the top level. The module display.v was written by Nathan Ickes to control the hexadecimal display on the new labkit. Its input is a 640 bit number that includes a bit for each diode on the display. Our module displaymsg3.v creates this 640bit number. Normally, the display shows CURRENT and then the time in hours, minutes and seconds. If the set alarm switch is up, then the display shows “ALARM” and then the alarm time in hours, minutes and seconds. If one of the alarm functions is being adjusted, then the name of that function and its delta time will be displayed. The delta time is a 6 bit number in sign magnitude format that indicates the relative time from the alarm time that that alarm function event occurs. To aid in reading, the displaymsg module, I wrote a submodule that takes in a six bit number and outputs the forty bits for a tens digit and the forty bits for a ones digit of that number in decimal notation for the display. This module is instantiated several times to find the tens and ones digits of the 3 clock and alarm times (hr, min, sec) and to find the tens and ones digit of the selected function’s delta time (min). Since the alarm functions are often activated before the alarm time, a minus sign is displayed so that the user knows that, for example, the music will start at alarm_time **-** 10 minutes. If the TA mode is selected by pushing button 1, HACK THE PLANET is displayed (credit suggestion by Nathan Ackerman). A yet undiscovered bug requires the 640 bits sent to display.v to be used somewhere else or the display would not light up correctly, so 2 bits were output to the labkit’s LEDs. The decision tree for what to display is shown in Figure 8.

FUNCTIONS

LIGHT MODULE JS

The light module is responsible for turning on a light upon activation by the alarm module, gradually increasing the light intensity from zero to max intensity over the time period starting at the light alarm time and ending five minutes after the main alarm time. The light module gradually increases the light intensity using a duty-ratio controlled square wave (changing logic levels). The duty ratio is set using a counter. The counter counts from zero up and is reset to zero each time it reaches a six bit max count provided by the alarm module and equal to the main alarm time plus five minutes minus the light alarm time, in minutes. On each clock cycle, the current count is incremented and compared to another six bit input from the alarm module equal to the current clock time minus the light alarm time. If the count is greater than this six bit input, than the square wave is low, otherwise it is high. In this way, the duty ratio, and thus the light intensity, will increase from zero to max as the clock time increases from the light alarm time to five minutes past the main alarm time.

MUSIC MODULE JS

The music module is responsible for playing mp3 files upon activation by the alarm module. The music module accomplishes mp3 playback via an ST Microelectronics mp3 decoder chip, model STA013. This chip is configured using the I^2C protocol. It takes in mp3 file data serially and provides serial digital audio output. The chip is capable of automatically detecting mp3 bit rate and sampling information, and it can automatically distinguish between mp3 file metadata and actual song data. It is also capable of independently controlling a DAC, but we have chosen to integrate the combine the digital audio output with the buzzer and voice memo audio signals. Once started, the music module performs two sequential operations. First, it initializes the mp3 decoder by writing to various chip addresses with data from a ROM instantiated in the FPGA. This initialization data configures a variety of decoder chip parameters such as PLL settings and the output oversampling rate. Second, after proper configuration, the music module begins sending mp3 file data until either the file ends or the module is stopped by the alarm module. Luckily, the I^2C signals are seperate from the mp3 data input signals, allowing the mp3 decoder communications to be modularized.

Music Initialization Submodule

The initialization submodule actually consists of a heirarchy of three submodules in order to simplify the complexity of the I^2C protocol. The lowest layer submodule in this heirarchy handles the I^2C transaction primitives for 'start', 'stop', 'send a byte', and 'receive a byte'. This submodule performs one of these primitives when provided with a start signal and, if appropriate, the byte to be sent from the next higher submodule. The primitives submodule will also leave a received byte for the next higher module if the primitive being performed is a receive. All primitives are performed using a two bit bus, one bit being used to set a data value and the other signal being used to clock that data value into either the decoder chip or the controller. The controller always maintains

control of the clock signal, but the two share control of the data signal. When a data bit is to be sent, it is simply placed on the data signal while the clock signal is low and then 'clocked in'. The only time the data signal falls or rises while the clock signal is high is during a start or stop operation, respectively.
The next higher module in the hierarchy handles the atomic operations of reading or writing a byte to a given address. This submodule performs on of these operations when provided with a start signal, an decoder chip address, and a write byte if the operation is a write. A read operation will leave a read byte for the next higher module. A read operation consists of the following sequence of I^2C primitives: a start, a sent byte indicating that the following primitive will be a send, a sent byte equal to the chip address to be read from, a stop, a second start, a sent byte indicating that the following primitive will be a receive, a received byte stored in the chip at the given address, and a final stop. A write operation consists of the following sequence of I^2C primitives: a start, a sent byte indicating that the following primitive will be a send, a sent byte equal to the chip address that will be written to, a sent byte with the data to be written at the given address, and a final stop. This read/write submodule is started by the top initialization submodule.
The top initialization submodule first checks for chip presence by reading from a read-only chip address and verifying a known data value stored there. The submodule then pulls some 2000+ address-data pairs from a ROM and writes the data to the decoder chip using the read/write submodule. Finally, the top initialization submodule writes a start value to a defined 'chip start' address, after which point the mp3 decoder chip raises a data request signal indicating that it is ready to start receiving mp3 data on the mp3 data signals. The top initialization submodule checks for this data request signal and, when received, drops its busy signal indicating to the music module that initialization is complete.

Music Data Submodule

The music data submodule handles sending mp3 file data serially to the mp3 decoder chip. The communications protocol used to send the mp3 data is quite simple relative to the I^2C protocol. The mp3 data bus consists of three one-way signals: a data request signal controlled by the decoder chip and two signals for data and clock provided by the controller, the FPGA in this case. When the mp3 decoder chip is ready for a data bit, it raises the data request line. Music data submodule then puts a bit on the data signal taken from a ROM instantiated in the FPGA, raises and then lowers the clock signal to latch the mp3 data bit into the decoder chip, and then waits for the next positive going data request signal.

BUZZER EF

The buzzer module uses a simple counter to generate an eight bit saw-tooth audio signal. It is implemented using an FSM that exits the IDLE state once it receives a start command from the alarm module. A saw-tooth signal was chosen for its irritating quality and the fundamental frequency was chosen to be about 440Hz since I knew for sure that this would be within an audio range. The 8 bit buzzer sound is sent to an audio module which is a simple FSM that adds the inputs to send to a DAC. This is shown in Figures 9 and 10.

VOICE MEMO EF

The voice module is an FSM with two major paths. The state is normally IDLE, but heads down the record path when it gets the signal to start recording (button 2). An internal SRAM is built, based on the 6.111 labkit website, from a very large register. On the record path, the address is incremented for about 7 seconds. The data stored during this time comes from an A/D converter that looks at the amplified signal from an unpowered microphone. The state machine for this module is shown in Figure 11.

Audio Module

The file audio.v added the signals from the buzzer and the voice memo together as shown in Figure 12. It also controlled the D/A converter. The D/A conversion of the voice signal did not accurately reflect the recorded analog signal at the time of this report, however, some sound was output on top of the buzzer sound, so portions of this module did work

DESIGN METHODOLOGY / DECISIONS / TRADE OFFS JS

Overall, our system functionality lent itself to a very modular design, and we focused on following this natural modularity at all levels of our design rather than try to force independent functionalities into the same module or state machine. This modularity had many benefits and few detractors. Major benefits included easier debugging, easier division of labor, and easier system changes due to the fact that the internal workings of a module could often be changed without affecting other modules so long as the module interface remained constant. The detractors included the additional code overhead from accomplishing the same functionality with more modules and the extra time involved in planning out robust module interfaces in the early design stages.

We decided to implement a 24 hour clock and our own time ALU to be used with a sectioned 18 bit time value format. The method was chosen over others such as having to deal with AM/PM or conversions between absolute seconds and a suitable display format. Overall, we had no problems with this design decision and found it to be of greater convenience in almost all implementation instances.

Initially, we had planned to implement the gradually increasing light functionality via a floating point division operation, but using a simple varied counting system proved to be surprisingly elegant and completely successful given our requirements.

Early on, the decision was made to rely on an mp3 decoder chip for mp3 playback rather than try to learn the mp3 decoding algorithm for FPGA implementation. It was assumed that this would provide a significant system simplification both in terms of implementation time and complexity. In the end, the I^2C protocol proved to be very complicated to implement, and, even after implementing it properly, we were unable to get the decoder chip to begin requesting mp3 data following the configuration phase. Without sufficient knowledge of the mp3 encoding complexity, it is difficult to say whether this option would have proved successful, but I think we would take a closer look at it if we had to start back at the beginning.

TESTING / SIMULATIONS

Most modules and submodules were simulated with testbenches in either ModelSim or Icarus Verilog prior to implementation. Exceptions to this generality included some tests involving hardware external to the FPGA, such as the mp3 decoder chip and DACs, due to the fact that extensive software simulation would have involved the possibly difficult task of simulating the response of the external hardware. Overall, however, the extensive software simulations that were performed contributed significantly to the lack of major bugs once the various modules were integrated together.

The mp3 decoder chip debugging was facilitated primarily using a logic analyzer. In the end, this tool proved essential to getting the I^2C communications functioning properly. Initially, however, the analyzer caused additional confusion because it was acting as a pull down on signals being monitored which were tri-stated at times.

The display functionality was mostly tested by Xilinx compilation and our observing the results. Fixing the elusive bug by sending two bits to the LEDs was an unusual find, but it worked.

CONCLUSION EF

We built an alarm clock prototype for the pleasant waking experience envisioned in beds across the world. An accurate clock keeps track of the current time and a display shows this information or other relevant information, like the time of the alarm and the times for each of the alarm functions. All six of these times can be adjusted separately. The alarm module accurately signals each function when to start. A light LED increases slowing to full power and an annoying buzzer sounds until the alarm is turned off. Significant steps were taken towards the full implementation of an MP3 player and a voice memo system. Most importantly, we learned so much that the professor and TAs gave us a great grade on the project.

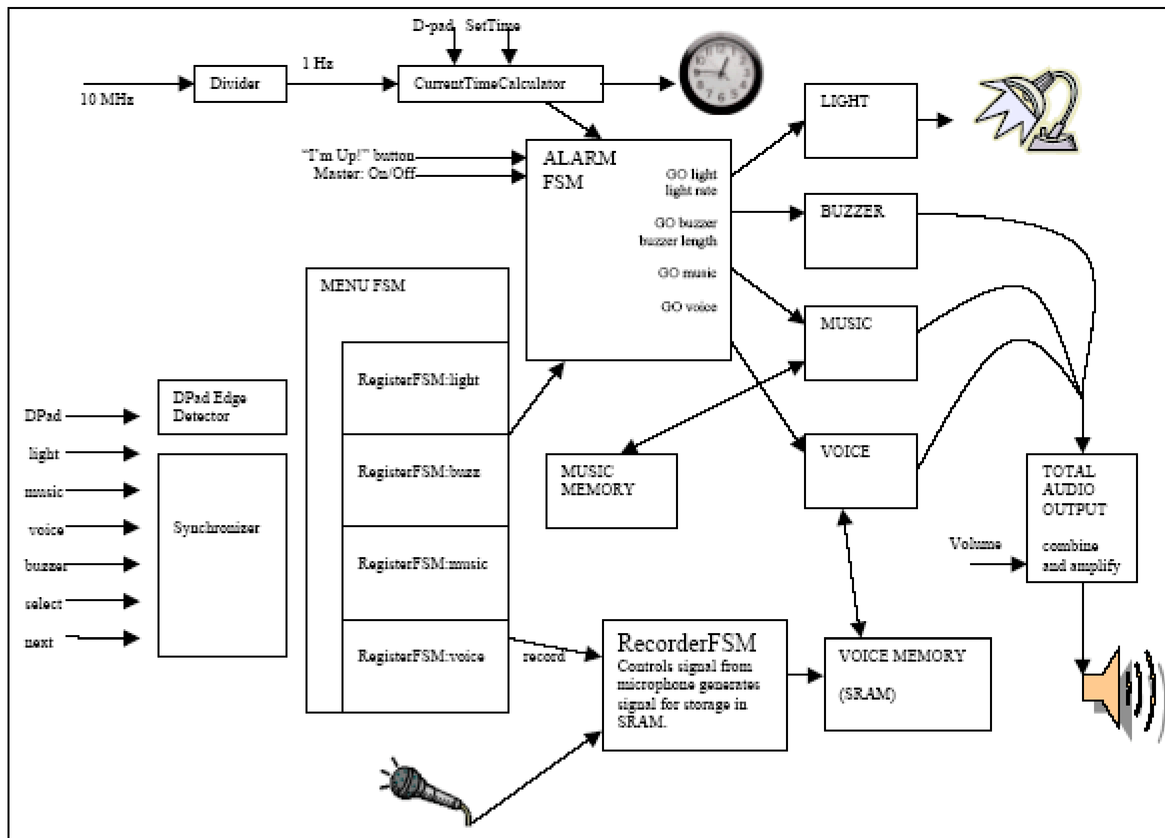

Figure 1: General Block Diagram

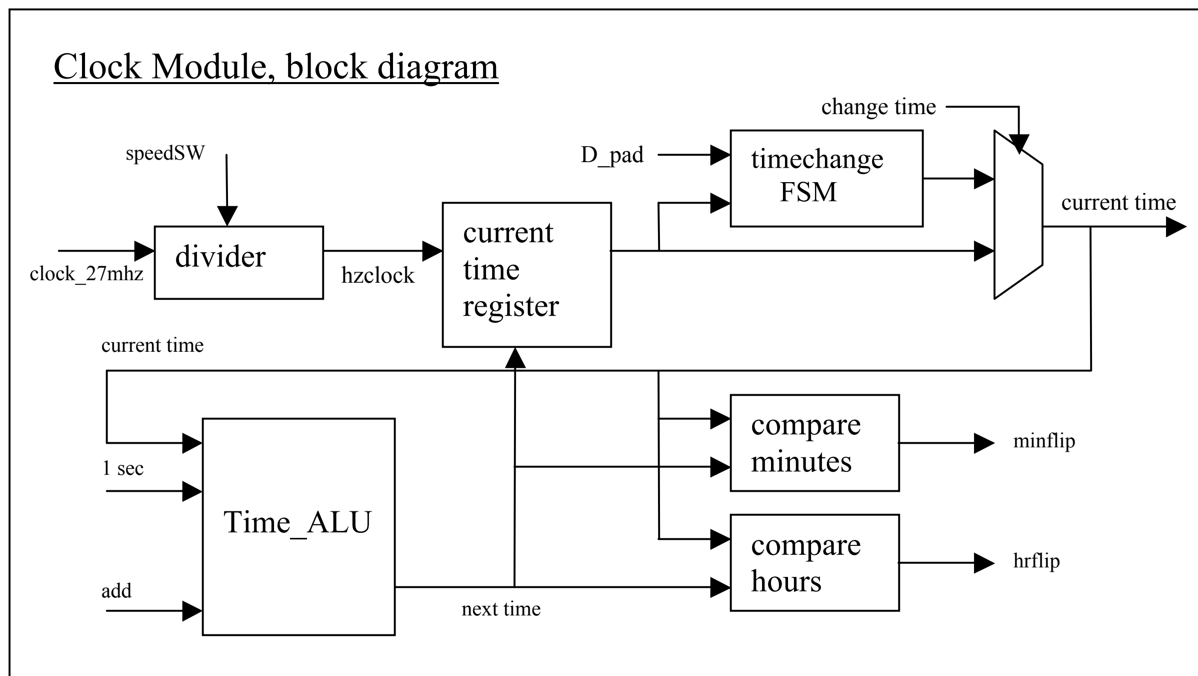

Figure 2: Clock Module, Block Diagram

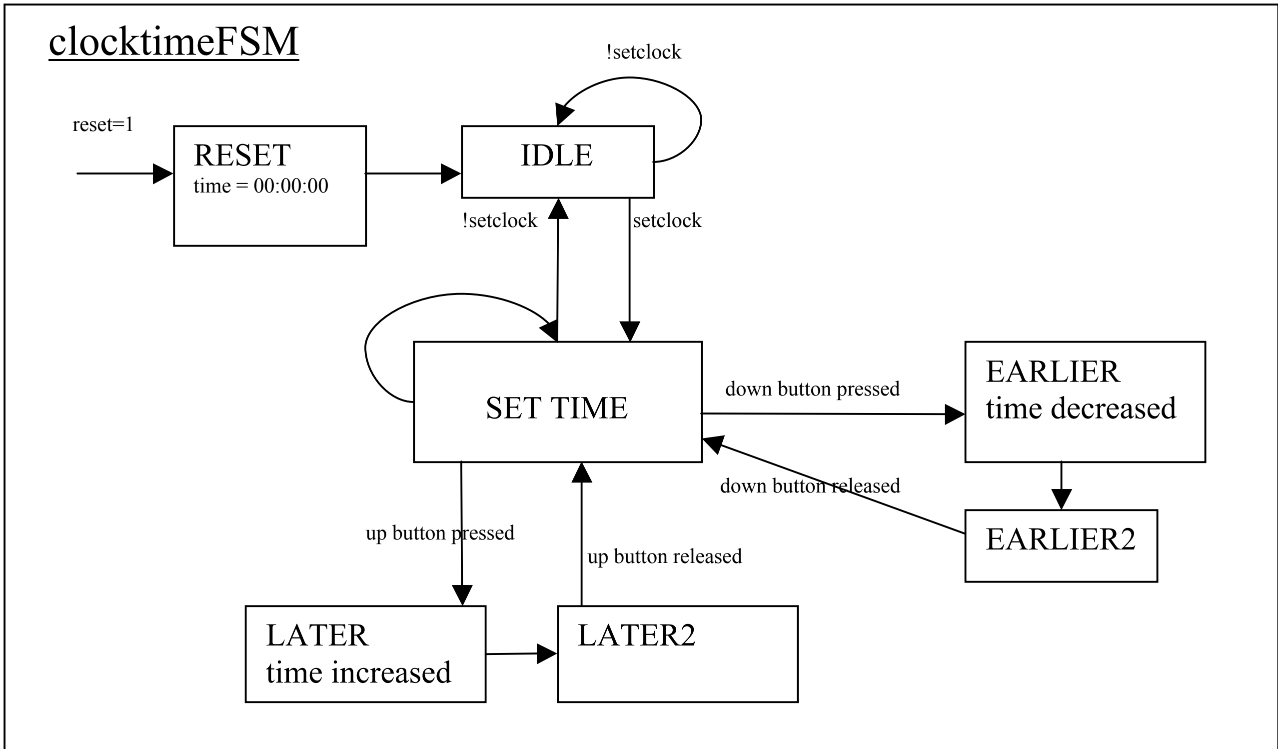

Figure 3: Clock Time Change, State Diagram

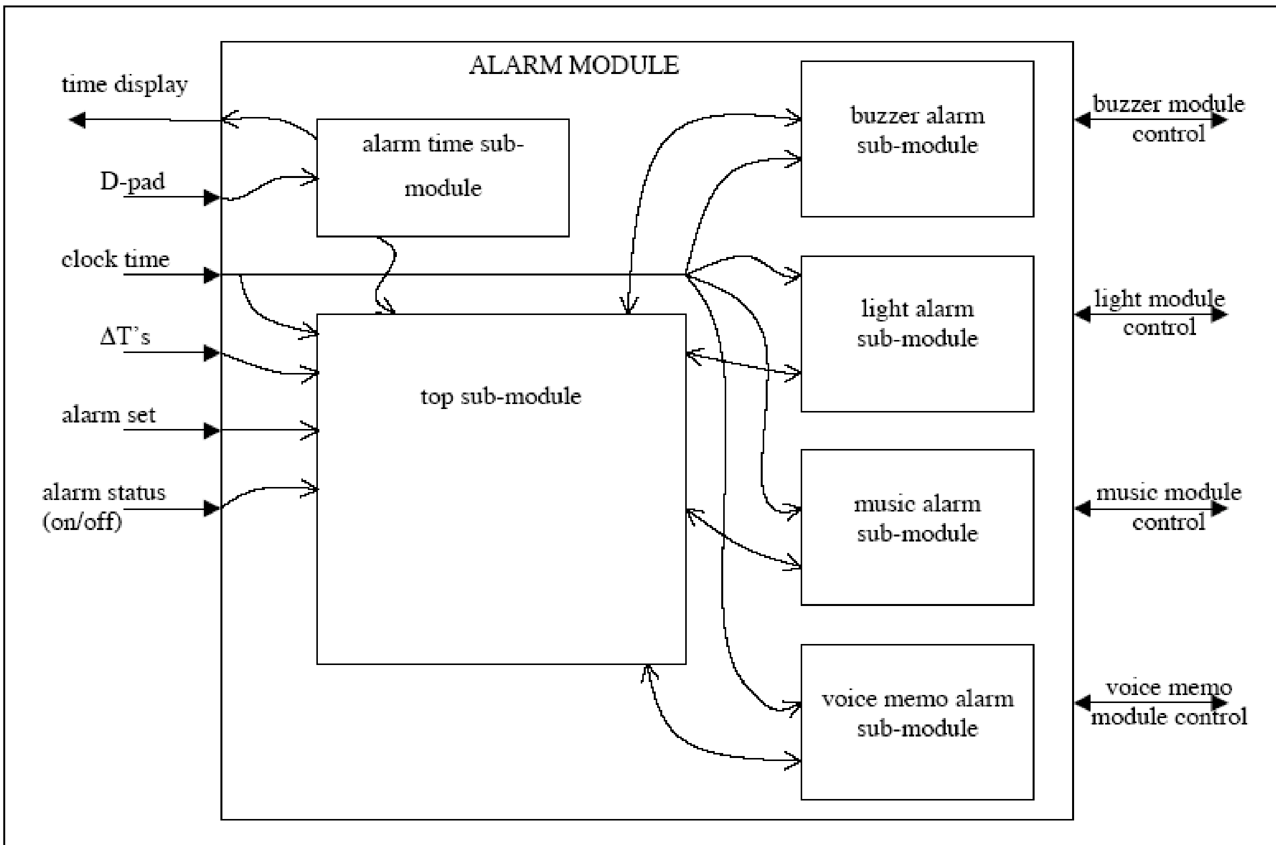

Figure 4: Block Diagram of Alarm Module

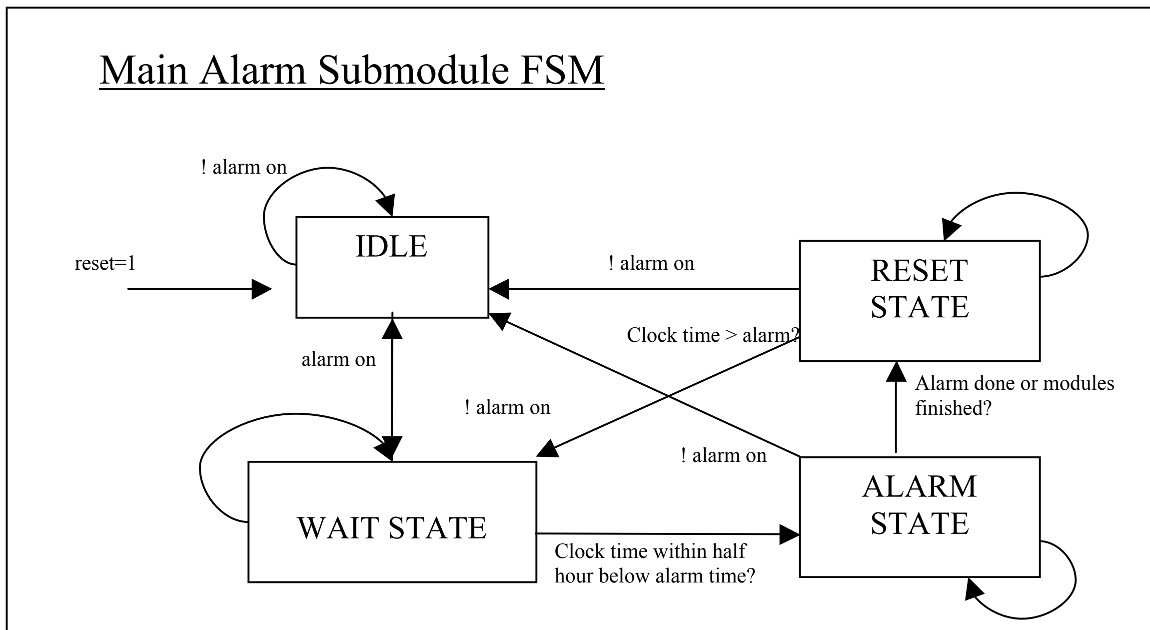

Figure 5: Main Alarm Submodule FSM, State Diagram

Menu Module, block diagram

clock
D_pad
reset
lightSW
buzzerSW
musicSW
voiceSW
MENU
CHOOSE
set light
Light: MenuFSM
light _t
set buzzer
Buzzer: MenuFSM
buzz _t
set music
Music: MenuFSM
music _t
set voice
Voice: MenuFSM
voice _t
set feature
feature_number
4x1 Mux
earlytime

Figure 6: Block Diagram of Menu Module

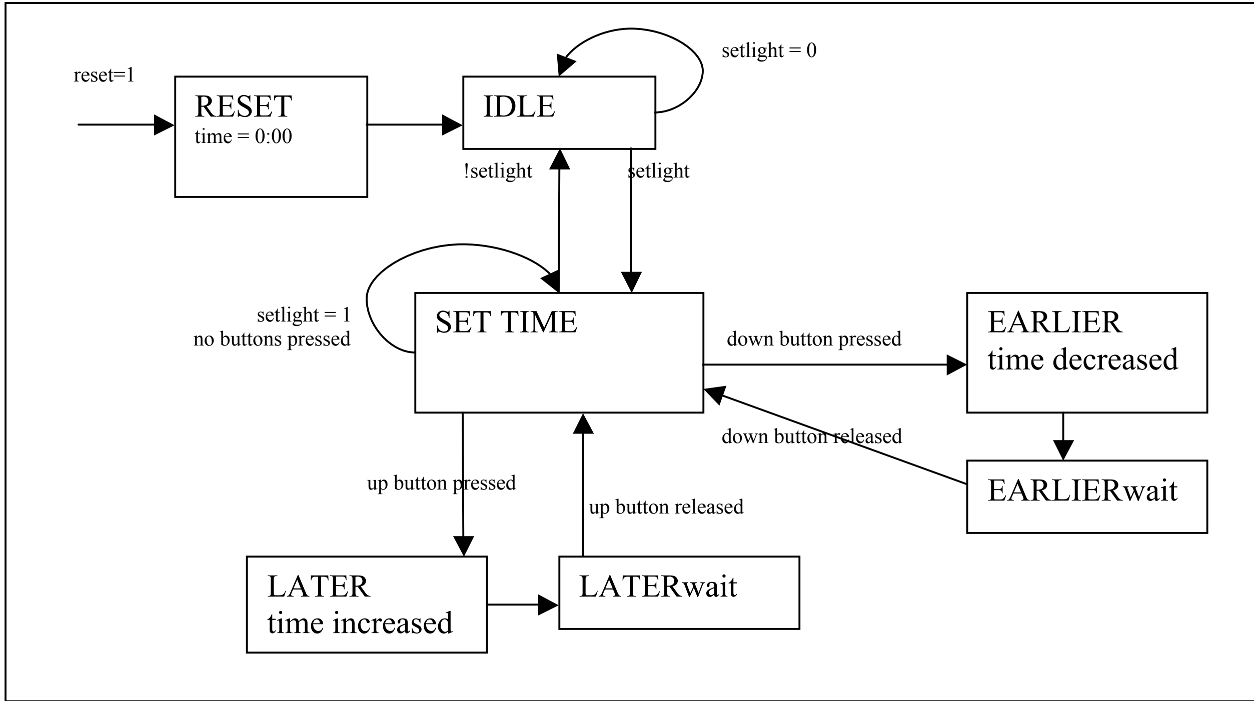

Figure 7: Small Menu FSM, State Diagram illustration for Light instantiation

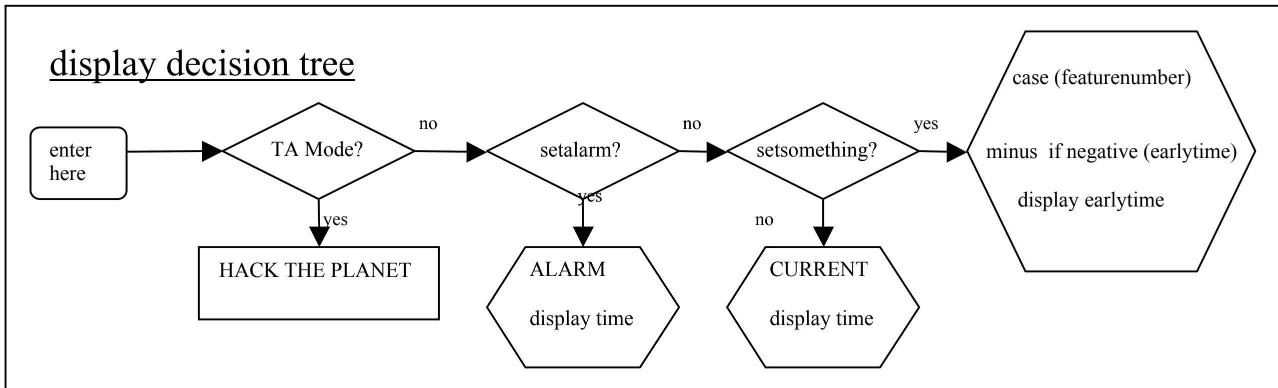

Figure 8: Display Decision Tree

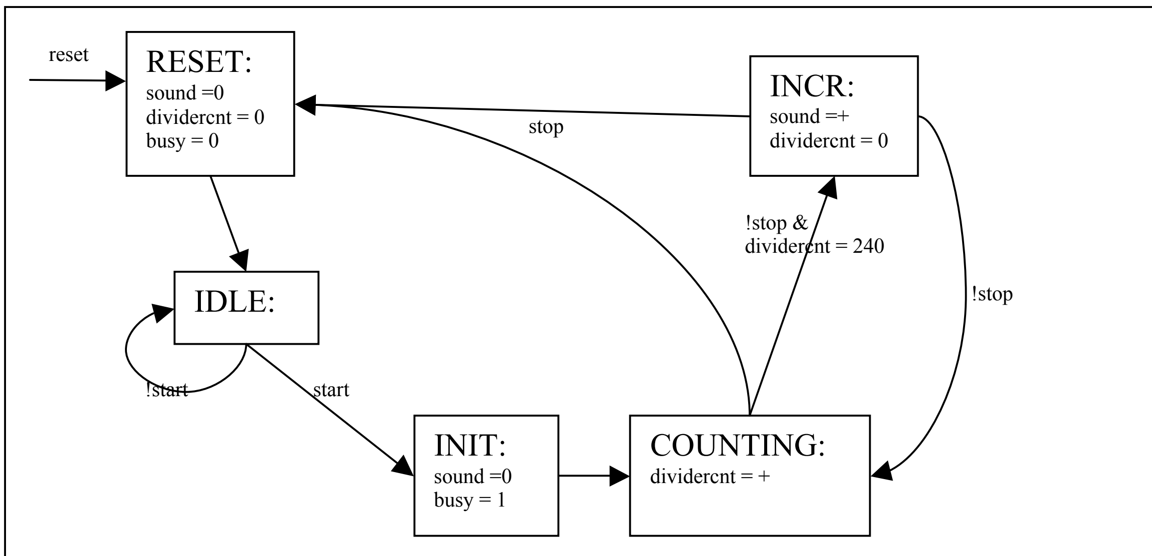

Figure 9: Buzzer Module FSM

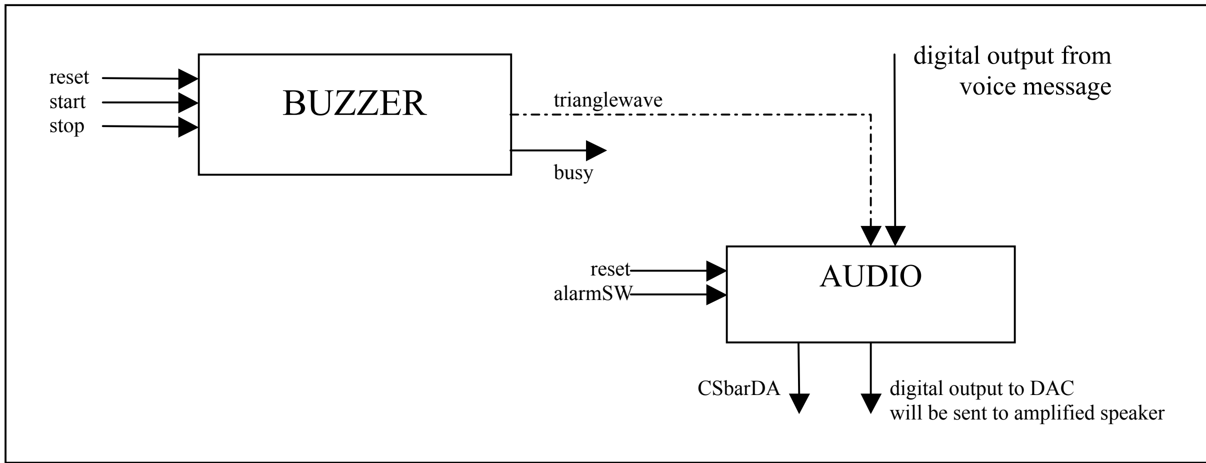

Figure 10: Buzzer and Audio modules

RECORD:
rcnt = +
SRAMaddr = +
RorWbar = 1
CSbarAD = 1
rcnt = 28
RorWbar = 0
CSbarAD = 0
SRAMWriteEnable =0
CSbarAD = 1
SRAMWriteEnable =1
SRAMdatain = soundin
rcnt = 12
RorWbar =1
CSbarAD = 0
WAITRECORD:
rcnt = +
rcnt =1350
CONVERSION WAIT:
rcnt = 0
SRAMaddr = 100K
INITRECORD:
rcnt = 13
SRAMaddr = 0
busy = 1
!start &
record
reset
RESET
busy =0
IDLE:
busy =0
stop
start
SRAMaddr = 100K
SRAMaddr /= 100K
PLAY2:
soundout = SRAMout
SRAMaddr = +
INITPLAY:
SRAMWriteEnable =0
rcnt = 1350
SRAMaddr = 0
busy = 1
WAITPLAY:
rcnt = +
internalcnt = 0
rcnt = 1350
PLAY1:
rcnt = 0
internalcnt = +

Figure 11: Voice Module FSM

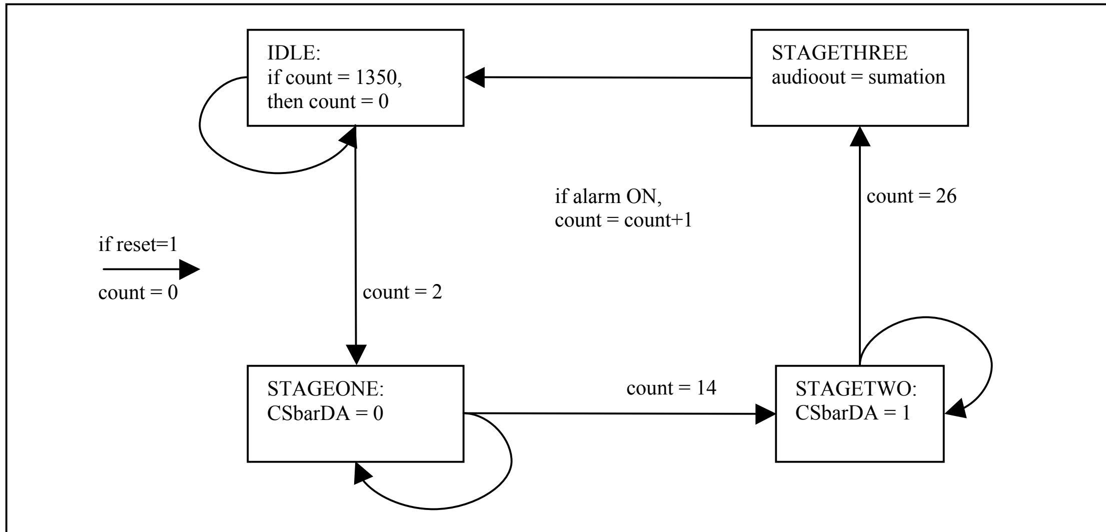

Figure 12: Audio FSM, State Diagram